Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES MATHÉMATIQUES À FINALITÉ SPÉCIALISÉE EN DATA SCIENCE

Réglage de la taille de la population de l'algorithme génétique au sein de l'optimisation assistée par modèles de substitution présent dans le logiciel Minamo

Derlet, Youri

Award date:
2019

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. May. 2023

Université de Namur

Faculté des Sciences

Réglage de la taille de la population de l'algorithme génétique au sein de l'optimisation assistée par modèles de substitution présent dans le logiciel Minamo

**Mémoire présenté pour l'obtention
du grade académique de master en
sciences mathématiques, à finalité spécialisée en Data Sciences**

Youri DERLET
Juin 2019

Université de Namur

Faculté des Sciences

Réglage de la taille de la population de l'algorithme génétique au sein de l'optimisation assistée par modèles de substitution présent dans le logiciel Minamo

Promoteur : T. CARLETTI

Co-promotrice : C. BEAUTHIER

Mémoire présenté pour l'obtention du grade académique de master en sciences mathématiques, à finalité spécialisée en Data Sciences

Youri DERLET

Juin 2019

Remerciements

D'abord, je tiens personnellement à remercier mon promoteur T. CARLETTI et ma co-promotrice C. BEAUTHIER avec qui j'ai pu réaliser mon mémoire durant un an et demi. Je tiens à reconnaître leur disponibilité et leurs nombreux conseils, ainsi que leur relecture consciencieuse de ce mémoire. De même, je remercie les membres de l'équipe Minamo de Cenaero qui m'ont aussi conseillé.

Je remercie également mes parents qui m'ont permis de réaliser des études dans le Département de Mathématique de la Faculté des Sciences de l'Université de Namur. Merci pour leur soutient tout au long de mes études !

Pour finir, mes derniers remerciements, mais pas des moindres, sont adressés au reste de ma famille et à mes proches. Je remercie personnellement Jelena qui m'a accompagné et soutenu durant mes études et qui a également relu consciencieusement ce mémoire.

Résumé

Ce mémoire a été réalisé en collaboration avec le centre de recherche Cenaero, qui a développé un logiciel d'optimisation multi-disciplinaire (Minamo), basé principalement sur un algorithme génétique, abrégé par AG et aussi dénommé algorithme évolutionnaire. Un tel algorithme est une méthode heuristique pour résoudre des problèmes d'optimisation, auxquels le monde industriel fait face quotidiennement. Cette méthode est inspirée de l'évolution darwinienne : elle fait évoluer une population de points en les croisant et en les mutant.

Au sein de Minamo, cet algorithme peut être intégré dans une boucle principale, qui permet de faire évoluer un méta-modèle approximant la véritable fonction objectif lorsque l'évaluation de celle-ci est coûteuse en temps de calcul. Cette méthodologie est alors appelée *optimisation assistée par modèles de substitution*. A chaque étape de cette approche, l'AG minimise le méta-modèle.

Le but de ce mémoire est de contrôler la taille de la population de l'AG au cours des itérations. Actuellement, cette taille est figée pour toutes les itérations à une unique valeur. Bien entendu, un tel contrôle a pour mission, d'une part, de réduire la population pour converger vers un minimum et, d'autre part, de l'augmenter pour parcourir d'avantage l'espace de conception du problème. De plus, le contrôle de la taille de la population a comme objectif de diminuer le nombre d'évaluations de fonctions afin de gagner du temps d'exécution, sans détériorer les solutions heuristiques par rapport à Minamo. C'est pour cela que le temps d'exécution est utilisé dans ce mémoire pour comparer les méthodes étudiées et Minamo. Le défi pour l'implémentation est que cette réduction d'évaluations de fonctions soit également bénéfique pour les exécutions de l'AG intégré dans la boucle d'optimisation assistée par modèles de substitution. En effet, l'évaluation de fonctions sur les méta-modèles est plus rapide que l'évaluation des véritables fonctions objectifs. Ainsi, la réduction du nombre d'individus doit être rentabilisée pour avoir effectivement une réduction du temps d'exécution sans détérioration de la recherche d'optima.

Trois méthodes ont été implémentées dans Minamo : *Saw-Tooth*, *GAVaPS* (pour l'acronyme *Genetic Algorithm with Varying Population Size*) et *FiScIS-EA* (pour *Fitness Scaled Individual Survival Evolutionary Algorithm*). En plus de celles-ci, nous avons apporté une amélioration pour la dernière. En effet, nous avons modifié la formule originale qui calculait la probabilité de survie. Ceci est bénéfique par rapport à la méthode originale.

Abstract

This master thesis was carried out in collaboration with Cenaero, a research center which developed a multi-disciplinary optimization software (Minamo). This latter is based mainly on an evolutionary algorithm (*EA*) which is a heuristic method for solving optimization problems. All theses problems occur in the industrial world. This method is inspired by Darwinian evolution. Indeed, it makes a population of points evolves by crossovers and mutations operators.

Within Minamo, this algorithm can be integrated into a main loop. This one develops a metamodel which approximates the real objective function which can be computationally expensive. This methodology is called *Surrogate-Based Optimization* (*SBO*). At each step of this approach, the *EA* minimizes the meta-model.

The purpose of this master thesis is to control the population size of the *EA* during the iterations. Currently, this size is fixed for all iterations at a single value. Of course, this control must reduce the population to find a minimum and increase it to cover the design space of the problem. In addition, the objective of this control is to reduce the number of function evaluations and to save time, without deteriorating heuristic solutions given by Minamo. This is why the execution time is used to compare the studied methods and Minamo in this master thesis. So, the challenge of the implementation is also to reduce the execution time for the *EA* integrated into a loop of the *SBO*. Indeed, evaluations on meta-models are faster than evaluations of real objective functions, without a deterioration of the search for optima. So, the reduction of the population size must be efficient to reduce the execution time.

Three methods have been implemented in Minamo : *Saw-Tooth*, *GAVaPS* (for *Genetic Algorithm with Varying Population Size*) and *FiScIS-EA* (for *Fitness Scaled Individual Survival Evolutionary Algorithm*). In addition, we have made improvements for the last method. Indeed, we have modified the original formula that calculated the probability of survival. This is advantageous compared to the original method.

Table des matières

Introduction

Ce mémoire est réalisé en collaboration avec le centre de recherche appliqué Cenaero, situé à Gosselies en Belgique. L'équipe avec laquelle nous travaillons est dédiée au développement du logiciel d'optimisation multi-discipline, Minamo, principalement basé sur un algorithme génétique (algorithme évolutionnaire, noté AG) assisté par méta-modèles (modèles de substitution).

Le *No Free Lunch Theorem* transcrit l'idée que, pour chaque algorithme, les valeurs de ses paramètres peuvent être optimisés pour un sous ensemble de problèmes d'optimisation au détriment des autres problèmes. Celui-ci stipule même que, sur tous les problèmes possibles et imaginables, la rapidité et l'efficacité de tous les algorithmes et de tous les paramètres se valent. Ce mémoire aura donc pour but d'améliorer le choix des valeurs, actuellement fixées, des différents paramètres au sein de l'algorithme génétique. Cette correction tentera au sein de l'exécution de déterminer les valeurs à utiliser.

Ce mémoire concerne l'algorithme génétique qui permet de solutionner de manière heuristique des problèmes d'optimisation. De manière synthétique cet algorithme est inspiré de la biologie. De cette manière, un point dans l'espace de résolution est appelé individu. Cette méthode utilise plusieurs individus qui représentent des solutions possibles au problème considéré. Ceux-ci forment une population qui au fil des itérations évolue par des combinaisons et mutations. Le but de ce mémoire sera de contrôler la taille de la population (donc le nombre d'individus) au fil de ces itérations. Ainsi, il ne sera plus utile de déterminer la taille de la population pour chaque problème d'optimisation, avant leur résolution. En effet, cela se fera au cours de l'évolution. L'intuition derrière une telle adaptation est de permettre d'apporter une variabilité, en ajoutant des individus, lors de la résolution quand la population d'individus est coincée dans un minimum local. Inversement, les méthodes doivent sélectionner de bons individus pour atteindre un minimum. Bien entendu, le but recherché au sein de ce mémoire est d'améliorer la vitesse de convergence sans détériorer la recherche de solution par rapport au code existant (Minamo). Ainsi nous comparerons toujours nos versions avec Minamo.

Après cette introduction, il est donc utile de commencer ce mémoire par le Chapitre 1 qui concerne la terminologie utilisée. Il rappellera d'abord ce que sont les problèmes d'optimisation avec les définitions de solutions locales et globales. Puis, nous développerons l'algorithme génétique avec ses différents paramètres et opérateurs. Nous décrirons également comment un individu est considéré meilleur qu'un autre, bon ou mauvais. Après avoir expliqué le *No Free Lunch Theorem*, nous définirons les différents types de réglages. Puis, dans le Chapitre 2, nous réaliserons une revue bibliographique des différents réglages existants dans la littérature sur les paramètres de l'algorithme génétique. Enfin, ce chapitre sera complété par un résumé des réglages que nous utiliserons dans ce mémoire. Le Chapitre 3 réalise une transition entre l'état de l'art réalisé et Minamo. Pour ce faire, une explication sur l'optimisation assistée par modèles de substitution sera développée. Nous y décrirons également les paramètres de l'algorithme génétique utilisé dans ce mémoire, avec ses différents paramètres fixés de base.

Après cela, dans trois chapitres différents, nous développerons trois réglages visant à faire évoluer la taille de la population de l'algorithme génétique durant les itérations. Premièrement, le

Chapitre 4 concernera le réglage *Saw-Tooth* [1] qui diminue linéairement la taille de la population avant de l'augmenter à sa taille originale. Cette action représente une dent, et la répétition de celle-ci représente une scie. Dans ce chapitre, nous décrirons la méthode et déterminerons les valeurs de ses paramètres au sein de Minamo, avec l'algorithme génétique pur puis avec celui-ci assisté par un méta-modèle. Deuxièmement, le Chapitre 5, concernant le réglage *GAVaPS* [2], aura la même structure. Cependant, celui-ci modifiera considérablement la philosophie cachée derrière l'algorithme génétique de Minamo. En effet, un individu ne va plus être supprimé à l'itération d'après mais pourra survivre durant un certain nombre d'itérations. Il aura ainsi une espérance de vie longue ou courte selon qu'il est bon ou non. Ensuite, dans le Chapitre 6, nous regarderons la méthode *FiScIS-EA* [3] qui ne possède aucun paramètre supplémentaire. Celle-ci laisse évoluer la population d'individus, en gardant les enfants et les parents à chaque itération, sans avoir de réel contrôle direct sur sa taille. En effet, chaque individu de la population d'enfants et de parents aura une chance de survivre pour l'itération suivante grâce à une probabilité proche de 1 s'il est bon ou proche de 0 s'il est mauvais. Nous allons également l'améliorer, en modifiant le calcul de la probabilité. Pour finir, nous réaliserons une seconde adaptation : nous avons fusionné la philosophie de *Saw-Tooth* à celle de *FiScIS-EA*. En effet, cette nouvelle méthode réalisera une décroissance sur la taille grâce à une probabilité de survie. Dans cette méthode, les enfants remplaceront les parents à chaque itération. Ces deux principes permettront une diminution du nombre d'individus. La population sera augmentée une fois que sa taille minimale est atteinte.

Pour terminer, le Chapitre 7 réalisera une comparaison des meilleures méthodes retenues et Minamo, avec l'AG pur ou avec l'AG assisté par modèles de substitution.

Glossaire

Dans ces deux pages, nous synthétisons les différentes notations utilisées au cours de ce mémoire dans le tableau suivant.

Variable(s)	**Définition(s)**
f	la fonction objectif
n_D	le nombre de dimension de la fonction f
n_I	le nombre de contraintes d'inégalités
n_E	le nombre de contraintes d'égalités
c_k	la $k^{\text{ième}}$ contrainte (où $k = 1, \ldots, n_I + n_E$)
t	l'itération courante de l'algorithme génétique (AG)
t_{max}	le nombre maximal d'itérations de l'AG
p_m	le taux (la probabilité) de mutation de l'AG
p_c	le taux (la probabilité) de croisement de l'AG
s	le nombre d'individus tirés aléatoirement pour la sélection par tournoi
x ou i	x et i représentent les individus de l'AG ; x correspond aux coordonnées, les chromosomes, d'un individu tandis que i représente son numéro (index) dans la population
$P(t)$ ou P_t	la population d'individus (ensemble d'individus) de l'AG à l'itération t
$E(t)$ ou E_t	l'ensemble des enfants générés dans l'AG à l'itération t
$\mu(t)$ ou μ_t	le nombre d'individus présents dans $P(t)$
$\overline{\mu}$	le nombre moyen d'individus présents dans la population au cours de toutes les itérations
μ_{min}	le nombre minimal d'individus autorisé dans la population
μ_{max}	le nombre maximal d'individus autorisé dans la population
D	l'écart entre $\overline{\mu}$ et μ_{max}
$\lambda(t)$ ou λ_t	le nombre d'individus présents dans $E(t)$
$\overline{T}$	l'itération courante de la boucle des modèles de substitution (SBO)
$\overline{t_{max}}$	le nombre maximal d'itérations de la boucle SBO
$\overline{n}$	le nombre de points présents dans la base de données (BD) de cette méthode
$\overline{x_k}$	le $k^{\text{ième}}$ point présent dans la BD de la boucle SBO (où $k = 1, \ldots, \overline{n}$)
$y(x)$	le modèle de substitution, défini pour tout x dans l'espace de conception
r	la base radiale
σ	un méta-paramètre
$h(r, \sigma)$	la fonction à base radiale
$x^\star$	les coordonnées du point qui minimise f selon les contraintes et qui est ainsi considéré comme la solution
$\mathcal{N}(x^\star)$	le voisinage du point $x^\star$

Variable(s)	**Définition(s)**
$fit(i)$ ou fit_i	la valeur de fitness pour l'individu i
fit_{min} ou $fit_{min}(t)$	la valeur de fitness minimale, parmi la valeur de fitness de chaque individu de la population $P(t)$
fit_{max} ou $fit_{max}(t)$	la valeur de fitness maximale, parmi la valeur de fitness de chaque individu de la population $P(t)$
fit_{moy} ou $fit_{moy}(t)$	la valeur de fitness moyenne, en considérant la valeur de fitness de chaque individu de la population $P(t)$
fit_{med} ou $fit_{med}(t)$	la valeur de fitness médiane, parmi la valeur de fitness de chaque individu de la population $P(t)$
$fit_{min,abs}$ ou $fit_{min,abs}(t)$	la valeur de fitness minimale (absolue), parmi la valeur de fitness de chaque individu des populations $P(\tilde{t})$ où $\tilde{t} < t$
$fit_{max,abs}$ ou $fit_{max,abs}(t)$	la valeur de fitness maximale (absolue), parmi la valeur de fitness de chaque individu des populations $P(\tilde{t})$ où $\tilde{t} < t$
η	la taille du pas dans la mutation de Gauss
m	la valeur moyenne des individus pour la mutation de Gauss
C	la matrice de covariance pour la mutation de Gauss
$\Phi(x)$	la distribution/mutation de Gauss sur l'individu x
a	l'algorithme a considéré
$\mathcal{X}$	le domaine du problème
$\mathcal{Y}$	l'ensemble image du problème
$d_k^x(k)$	la $k^{\text{ième}}$ valeur parcourue dans $\mathcal{X}$
$d_k^y(k)$	la $k^{\text{ième}}$ valeur parcourue dans $\mathcal{Y}$
$\mathcal{F}$	l'ensemble des fonctions objectifs
q	un nombre tiré aléatoirement entre zéro et un
γ	le taux d'apprentissage
$EV(i)$ ou EV_i	l'espérance de vie (restante) de l'individu i
EV_{min}	l'espérance de vie minimale (restante) qui peut être attribuée
EV_{max}	l'espérance de vie maximale (restante) qui peut être attribuée
φ	la constante calculée $\frac{1}{2}(EV_{max} - EV_{min})$
ρ	le taux de reproduction
I_{sup}	ensemble d'indices d'individus à supprimer
d_t	la distance à la solution à l'itération t
t_w	le temps d'absorption
t_l	le temps additionnel (de latence)
θ	un premier coefficient de proportionnalité pour t_w et t_l
β	un second coefficient de proportionnalité
ν	un troisième coefficient de proportionnalité
op_k	le $k^{\text{ième}}$ opérateur faisant partie d'un ensemble d'opérateurs
δ_k	le delta local (poids) pour l'opérateur k
p_e	la probabilité pour choisir un opérateur donné
D	l'amplitude de la taille de la population (entre $\overline{\mu}$ et μ_{max})
T	le nombre d'itérations pour réaliser une période dans *Saw-Tooth*
$\overline{T}$	le nombre de périodes souhaitées dans *Saw-Tooth*
q	nombre aléatoire tiré entre zéro et un
n_{stat}	le nombre d'exécutions de l'algorithme avec n_{stat} graines aléatoires différentes
δ	l'écart-type de l'évolution de la recherche sur n_{stat} exécutions différentes

Chapitre 1

Terminologie

Dans ce chapitre, nous allons rappeler ce que sont les problèmes d'optimisation avec les définitions de solutions locales et globales. Ensuite, nous citerons différentes méta-heuristiques qui permettent de résoudre de tels problèmes. Puis nous développerons complètement l'algorithme génétique, en assistant sur le fait que ses opérations comportent des choix à faire. En effet, différentes méthodes permettent de les coder, chacune d'elles comportant plus ou moins de paramètres. Après cela, nous énoncerons le *No Free Lunch Theorem* qui assistera sur le fait de régler les paramètres de l'algorithme. La dernière section concernera la terminologie de tels réglages.

1.1 Problèmes d'optimisation

Le problème classique en optimisation est de minimiser une fonction objectif f, à valeur réelle et définie sur $\mathbb{R}^{n_D}$ où n_D est le nombre de dimensions. Chaque problème comporte des contraintes d'égalité (au nombre de n_E) et d'inégalité (n_I). Un point qui respecte toutes les contraintes du problème est caractérisé comme étant *faisable*. Remarquons que le domaine de f, si f doit être définie sur des intervalles inclus à $\mathbb{R}$, disparaît dans les contraintes du problème. Remarquons aussi que tout problème qui doit être maximisé peut être ramené à un problème de minimisation en prenant $-f(x_1, x_2, \ldots, x_{n_D})$ pour toutes valeurs $(x_1, x_2, \ldots, x_{n_D})$ qui respectent les contraintes. La définition d'un tel problème est donc la suivante :

$$\begin{cases} \min\limits_{(x_1,x_2,\ldots,x_{n_D})\in\mathbb{R}^n} & f(x_1, x_2, \ldots, x_{n_D}) \\ \\ s.c. & c_k(x_1, \ldots, x_{n_D}) = 0, \quad \forall\ k = 1, \ldots, n_E \ \text{ et } \ \forall\ (x_1, \ldots, x_{n_D}) \in \mathbb{R}^{n_D} \\ & c_k(x_1, \ldots, x_{n_D}) \geq 0, \quad \forall\ k = n_E + 1, \ldots, n_E + n_I \ \text{ et } \ \forall\ (x_1, \ldots, x_{n_D}) \in \mathbb{R}^{n_D}. \end{cases}$$

1.2 Solution locale ou globale

Dans notre mémoire, nous considérons uniquement les problèmes définis sur un domaine de définition continu (réel). Ainsi, une solution locale $x^\star \in \mathbb{R}^{n_D}$, qui respecte les contraintes, minimise la fonction objectif que dans un voisinage proche de celle-ci, noté $\mathcal{N}(x^\star)$. En d'autres termes, il faut que $f(x^\star) \leq f(x)$ pour tout $x \in \mathcal{N}(x^\star)$. Une solution locale stricte respecte une contrainte semblable, mais plus pointilleuse. En effet il faut que, pour tout $x \in \mathcal{N}(x^\star)$, $f(x^\star) < f(x)$. La notion de voisinage représente bien souvent une boule ouverte centrée en $x^\star$ dans le cas réel. La solution globale, quant à elle, ne nécessite plus une définition de voisinage. Pour l'optimisation classique, $x^\star \in \mathbb{R}^{n_D}$ est un minimum global (strict) si et seulement si il respecte les contraintes et si, pour tout $x \in \mathbb{R}^{n_D} \setminus \{x^\star\}$ qui respecte les contraintes, $f(x^\star) \leq f(x)$ (respectivement, $f(x^\star) < f(x)$). Dans ce mémoire, nous tenterons d'obtenir pour n'importe quel problème une solution globale, en évitant les pièges des solutions locales. Nous pouvons maintenant citer des méthodes heuristiques qui permettent de résoudre des problèmes d'optimisation de domaines continus.

1.3 Méthodes heuristiques

Les méthodes heuristiques sont une obligation pour résoudre des problèmes d'optimisation de la vie courante (problèmes opérationnels). En effet, ils sont nombreux à demander beaucoup de temps de calcul pour trouver une solution adéquate car ils comportent beaucoup de variables et car l'évaluation de la fonction objectif peut être longue. Citons par exemple la météorologie qui nécessite énormément de données en différentes altitudes, longitudes et latitudes, et ce à différents moments. De plus, dans de multiples disciplines, comme par exemple l'aviation (où une fois que l'avion est dans le ciel, il doit prendre des décisions rapidement) et les transports en commun (où des horaires doivent être recalculés au moindre retard), une solution proche de la véritable solution est satisfaisante. Celle-ci est appelée solution heuristique : elle minimise au mieux le problème opérationnel considéré.

Blum et Roli [4] présentent différentes méta-heuristiques pour l'optimisation combinatoire, qui peuvent être adaptés à l'optimisation classique (que nous faisons). Ils décrivent notamment deux classes de méthodes : la première est basée sur les trajectoires d'un seul point et la seconde sur des populations. La première comporte des méthodes pour rechercher un minimum local, comme l'amélioration itérative (*Iterative Improvement*), mais également des algorithmes comme le recuit-simulé (*Simulated Annealing*) et le *Tabu Search* qui tentent de rechercher des minima globaux. Les méthodes basées sur des populations permettent d'obtenir des minima locaux. Citons, par exemple, l'optimisation par colonies de fourmis (*Ant Colony Optimization*) et le calcul évolutif (*Evolutionary Algorithm*), aussi appelé **algorithme génétique** (*Genetic Algorithm*), que nous définirons complètement dans la section suivante (en détaillant ses opérations et ses paramètres).

Le recuit simulé est une méta-heuristique inspirée d'un procédé physique qui permet d'obtenir des propriétés sur différents métaux. En optimisation, ce processus nécessite un paramètre : la température. Elle permet d'éviter de rester coincé dans un minimum local. Le principal problème de cette méthode est que l'exécution reste bloquée dans des puits locaux. Il n'y pas de certitude concernant la solution trouvée. Cela est occasionné par le fait qu'il ne retient pas les points déjà abordés. L'algorithme heuristique suivant implémente un historique.

Le *Tabu Search* permet donc de s'échapper de puits locaux en utilisant un historique de recherche, sous forme de liste de points (la *Tabu List*), et en implémentant une stratégie pour explorer localement une solution potentielle. Son procédé permet de ne pas revenir sur un élément déjà abordé, du moins tant qu'il reste dans la liste.

L'optimisation par des colonies de fourmis utilise également une forme de mémorisation, celle que représente les phéromones que ces animaux déposent. Elle est particulièrement bien adaptée pour le problème du voyageur du commerce (*Traveling Salesman Problem*). Ce problème peut être adapté pour parcourir un domaine continu [5]. Dans leur article, Socha et Dorigo appuient sur le changement de la représentation des phéromones et le passage à une probabilité continue et non plus discrète.

Notons que ces algorithmes sont moins adaptés par rapport à l'algorithme génétique pour l'optimisation sur un domaine continu. A présent, nous allons l'expliquer en détaillant ses opérations et ses paramètres.

1.4 Algorithme génétique (AG)

L'AG est inspiré de l'évolution darwinienne. Pour traduire ce concept en programmation, Blum et Roli [4] rappellent qu'un individu représente les coordonnées d'un point dans l'espace de concep-

tion (ce sont ses caractéristiques). Chacune des variables représente un gène. Ces caractéristiques définissent le **génotype** de l'individu. Puis, à une itération t donnée, comme les individus forment une population d'une certaine taille μ_t, il est difficile de dire quel individu est le meilleur sur base des génotypes de tous les individus. C'est pour cela que ces gènes sont évalués par la fonction objectif du problème d'optimisation ou par une fonction d'évaluation permettant de mesurer la qualité d'un individu, appelée **fonction de fitness**. A l'inverse de la fonction objectif, qui doit être minimisée, la fonction de fitness doit être maximisée. Par exemple, elle peut être établie par l'inverse (ou même l'opposé) de la fonction objectif. La valeur d'une de ces fonctions d'un individu représente le **phénotype** de celui-ci. De plus, un individu est dit **meilleur** qu'un autre si sa fonction de fitness est plus élevée que celle du second.

Pour commencer l'AG, il faut générer une population de points évalués, ayant une certaine taille μ_0. Puis, Blum et Roli [4] répertorient trois opérations génétiques : le **croisement** (*Crossover*), la **mutation** et la sélection. Cette dernière opération sera considérée dans ce mémoire comme la **création de la nouvelle génération**. Notons que, dans la littérature, la sélection peut aussi apparaître afin de **choisir des individus pour le croisement**. Le croisement permet de fusionner plusieurs individus entre eux pour générer des nouveaux individus, appelés enfants, afin d'exploiter des régions locales de l'espace de faisabilité. La mutation d'un individu est appliquée sur certaines de ses caractéristiques, afin d'explorer l'espace de faisabilité. La sélection permet, quant à elle, de choisir les individus qui feront partie de la génération suivante. Ainsi, une population composée d'un certain nombre d'individus se voit appliquer ces trois opérations successivement. Une condition d'arrêt peut être déterminée afin de stopper ces itérations. Celle-ci doit tenir compte que certains points sont des minima locaux et que l'algorithme n'évolue plus pendant plusieurs générations. D'autre part, la condition d'arrêt peut correspondre au fait que le temps d'exécution imparti est écoulé, voire que le nombre d'itérations souhaitées est atteint. Une solution obtenue lors d'une exécution de l'AG n'est pas spécialement la même si l'AG est exécuté à nouveau, comme cet algorithme est une méthode heuristique. Détaillons ces étapes qui sont synthétisées dans la Figure 1.1.

Le croisement est réalisé à partir d'une certaine proportion de la population déterminée par le **taux de croisement** (aussi appelé probabilité de croisement). Celui-ci est défini par un réel compris entre zéro et un, dénoté par l'abréviation p_c. Si $p_c = 1$ alors tous les individus participeront par croisement à la création de la nouvelle génération. Au contraire, si $p_c \approx 0$ alors peu d'individus participeront au croisement. Typiquement, si $p_c \in]0, 1[$ alors il faut réduire la population de parents afin de ne garder que $100 \times p_c$ % de celle-ci. C'est la **sélection de parents**. Celle-ci peut être réalisée soit par une sélection précédant le croisement ou soit durant la création de groupes composés de futurs parents. Dans le premier cas, il existe différents opérateurs de sélection, comme ceux basés sur des tournois d'une certaine taille ou sur une roulette. Ces méthodes seront développées dans quelques paragraphes. Le nombre d'enfants par croisement est encore un autre paramètre. De plus, il existe différents opérateurs permettant de croiser des individus afin de donner des enfants. Par exemple, citons :

1. le croisement selon un nombre déterminé de points (Figure 1.2) ;
2. le croisement uniforme selon une probabilité (Figure 1.3).

En considérant, par exemple, qu'il n'y a qu'un enfant qui naîtra de deux parents, le premier consiste à sectionner des parties du génotype de chacun des parents pour créer celui de l'enfant, selon le nombre de points augmenté d'une unité. En effet, en coupant le génotype d'un individu en trois points, nous obtenons quatre groupes de gènes. La complétion de l'enfant est réalisée en alternant la provenance des gènes des parents. Ainsi, la première partie des gènes de l'enfant provient de la première section du premier parent. Ensuite, la seconde partie provient de la seconde section du second individu. Puis, nous recommençons avec le premier parent tout en alternant la provenance des groupes de gènes jusqu'à ce que l'enfant soit complété.

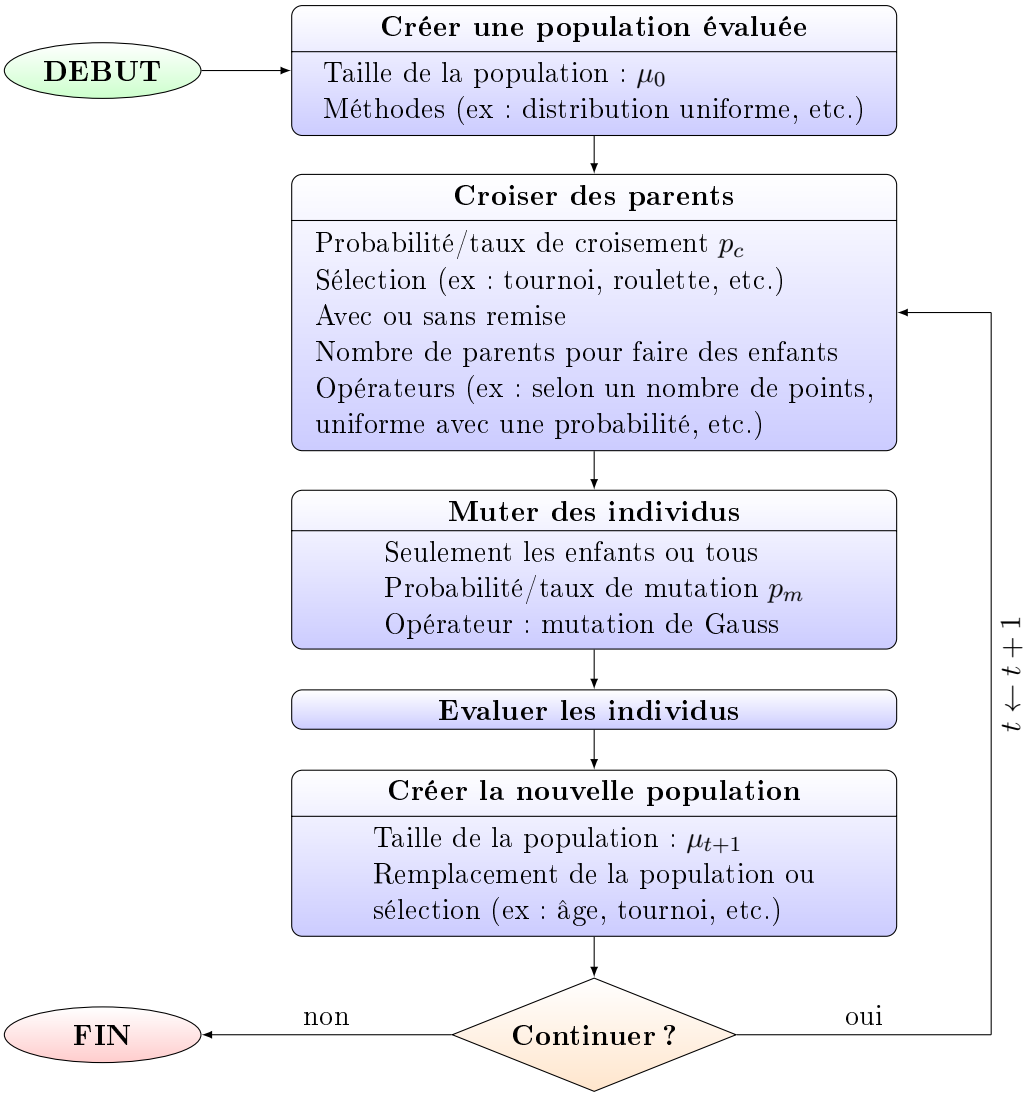


FIGURE 1.1 – Ce schéma représente les différentes opérations de l'algorithme génétique avec ses paramètres et des possibilités d'opérateurs.

Le deuxième opérateur sélectionne les gènes parmi ceux des deux individus. Considérons toujours dans cette explication que deux parents sont croisés pour créer un enfant (ce qui sera le cas dans ce mémoire). D'un point de vue algorithmique, pour chaque gène, un nombre aléatoire entre zéro et un est tiré, si celui-ci est plus petit que le taux déterminé alors le gène proviendra du premier parent (le meilleur), sinon le gène proviendra du second.

Parent 1 :	1.0	1.1	**1.2**	1.3	1.4	1.5	**1.6**	1.7	1.8	1.9
Parent 2 :	2.0	2.1	**2.2**	2.3	2.4	2.5	**2.6**	2.7	2.8	2.9
Enfant 1 :	1.0	1.1	**1.2**	2.3	2.4	2.5	**2.6**	1.7	1.8	1.9

FIGURE 1.2 – Ce schéma représente le croisement de deux parents pour donner un enfant, selon un nombre déterminé de points. Ici, deux points sont utilisés et sont représentés en gras. Chaque individu est formé de dix chromosomes.

Parent 1 :	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
Parent 2 :	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9
Enfant 1 :	1.0	1.1	1.2	1.3	2.4	1.5	1.6	1.7	2.8	1.9

FIGURE 1.3 – Ce schéma représente le croisement uniforme selon une probabilité de deux parents pour donner un enfant. Chaque individu est formé de dix chromosomes. Ici, un résultat d'un croisement réalisé avec une probabilité de 0.8 est donné à titre illustratif où le premier parent est le meilleur. Ainsi en moyenne, 80% des gènes proviennent du premier parent et, dans cet exemple, le cinquième et neuvième gêne proviennent du second parent.

Dans le cas réel, l'**opérateur de mutation** le plus connu est la mutation de Gauss. En considérant l'espace de recherche de la fonction à minimiser par $\mathbb{R}^{n_D}$, Eiben et al. [6] rappellent que, pour tout individu $x \in \mathbb{R}^{n_D}$:

$$\Phi(x) = \frac{exp\{-0.5(x-m)^T C^{-1}(x-m)\}}{\sqrt{(2\pi)^{n_D} \mid C \mid}}$$

où

1. m est une moyenne ;
2. C est une matrice de covariance (valant bien souvent l'identité) ;
3. $\mid C \mid$ est le déterminant de la matrice de covariance.

En d'autres mots, la distribution des individus $\Phi(x)$ dans l'espace de conception suit une distribution normale multivariée $\mathcal{N}(m, C)$. Ainsi, pour muter un individu x, nous pouvons utiliser la loi normale multivariée centrée en zéro, pour ajouter une perturbation à x, comme suit :

$$x \leftarrow x + \eta\, \mathcal{N}(0, C)$$

où η est **la taille d'un pas** (*Step Size*).

Le **taux de mutation**, dénoté p_m, n'est quant à lui pas une probabilité appliquée sur toute la population mais sur le génotype de chaque individu qui doit être muté. Ce taux représente donc le pourcentage des gènes qui devront être modifiés. Par exemple, pour un individu donné, si $p_m = 0$ alors aucun gène ne lui sera muté. Si $p_m = 1$ alors tout son génotype devrait être muté. D'un point de vue algorithmique, si $p_m \in]0, 1[$ alors, pour chacun de ses gènes, un nombre est tiré aléatoirement selon une loi uniforme entre zéro et un, et si celui-ci est plus petit que p_m alors le gène est muté. Ainsi le taux de mutation vaut un. De plus, la mutation n'est pas spécifiquement appliquée sur tous les individus. Elle pourrait être seulement appliquée sur les enfants, si nous savons que la génération suivante n'est composée que des enfants.

La **création de la génération suivante** se fait soit par remplacement de l'ancienne par les enfants mutés, soit par sélection d'individus parmi les parents et ses enfants. Cette sélection est réalisée, par exemple, en injectant une espérance de vie maximale et un âge à chaque individu, auquel cas si son âge dépasse son espérance de vie alors il décède et n'est donc pas répertorié dans la génération suivante. Nous verrons cette méthode dans la Section 2.1.1. Des méthodes de sélection plus traditionnelles seront développées dans le paragraphe suivant. Notons que la taille de la population suivante n'est pas spécialement égale à celle de l'actuelle.

La sélection de parents ou d'individus pour la génération suivante nécessite un opérateur, en voici trois :

1. la sélection par tournoi (*Tournament Selection*) ;
2. la sélection par roulette (*Roulette Wheel Selection*) ;
3. le classement linéaire (*Linear Ranking*).

Ils sont valables aussi bien pour générer la nouvelle génération que pour choisir les parents. Selon Miller et Goldberg [7], la sélection par tournoi est principalement utilisée pour trouver des parents comme elle choisit parmi les meilleurs individus. Nous expliquerons cette méthode pour créer l'ensemble des individus à croiser, mais une explication semblable peut être réalisée pour la génération de la nouvelle population. A une itération t donnée, parmi la population de taille n_t, s individus sont tirés aléatoirement. Puis survient une compétition entre ces individus, celui avec la meilleure valeur de fitness survit et est injecté dans l'ensemble de parents. Notons qu'il ne devra pas être ajouté à nouveau. Si cet ensemble n'est pas rempli, alors une compétition est à nouveau organisée. Sinon, le croisement est réalisé. En accord avec Pencheva et al. [8] qui citent Holland [9], la sélection par roulette permet quand même de sélectionner des individus qui n'ont pas une valeur de fitness avantageuse. En effet, la probabilité de choisir un individu i (pour qu'il devienne parent ou pour qu'il fasse partie de la population suivante) est calculée comme suit :

$$\frac{fit(i)}{\sum_{\substack{\text{individus } j \text{ qui} \\ \text{peuvent être} \\ \text{sélectionnés}}} fit(j)}$$

où $fit(i)$ est la valeur de fitness de l'individu i. D'un point de vue algorithmique, après avoir calculé les probabilités cumulées et tant que l'ensemble qui doit être généré n'est pas entièrement complet, un nombre aléatoire est tiré entre zéro et un, puis l'individu correspondant est ajouté à l'ensemble, s'il n'en fait pas déjà partie. Le classement linéaire se base sur la pression de sélection, notée SP pour *Selective Pressure* [10]. C'est la probabilité que le meilleur individu soit sélectionné en comparaison à la moyenne des probabilités de tous les autres individus. Pour cette méthode, celle-ci est fixée ou évolue durant l'exécution mais, pour une itération fixée t, elle est figée. Les individus sont ordonnés selon leur valeur de la fonction objectif : celui ayant la plus grande valeur est à la première position et celui ayant la plus petite est à la dernière position (n_t). Le calcul de la valeur de fitness, pour un individu i, est réalisé comme suit :

$$fit(i) = 2 - SP + 2(SP - 1)\left(\frac{i-1}{n_t - 1}\right).$$

Ainsi, les valeurs de fitness sont calculées par une décroissance linéaire et non plus selon la valeur même de la fonction objectif. Ceci donne encore un peu plus de chance à un mauvais individu d'être sélectionné. Cela permet ainsi d'augmenter la diversité de la population.

Notons que, par définition, l'AG ne tient pas compte des contraintes. Pour tenir compte de ces contraintes (quand elles ne sont pas respectées), une pénalité peut être introduite dans la fonction objectif. Cette pénalité doit être nulle si toutes les contraintes sont respectées. Deb [11] propose de remplacer la valeur de la fonction objectif, qui est ici à minimiser et non pas à maximiser, par une pénalité qui est également à minimiser. Ainsi, lorsqu'un tournoi entre deux individus est réalisé, trois possibilités sont envisagées.

1. Si les deux individus sont faisables, alors ceux-ci n'ont pas reçu de pénalité. Ainsi, le meilleur individu sera bien celui qui a sa valeur pour la fonction objectif la plus basse.
2. Si deux individus sont non-faisables, alors le meilleur individu sera celui qui respecte le plus les contraintes.
3. Pour finir, si un est faisable et l'autre non, alors le meilleur individu sélectionné sera le faisable.

Les grandes étapes de l'AG avec ses différents paramètres principaux sont synthétisées dans la Figure 1.1.

1.5 *No Free Lunch Theorem*

Le nom de ce théorème pourrait être traduit littéralement par "pas de repas gratuit", mais cela réfère en français plus à l'expression "On n'a rien sans rien". L'idée de ce théorème a pour but de prouver qu'un algorithme peut être super performant pour quelques fonctions objectifs et pitoyable pour la plupart des autres fonctions. Wolpert et Macready [12] ont donné ce nom à leur théorème pour cette raison et l'ont prouvé en annexe de leur article. Nous allons le citer mathématiquement. En considérant une fonction objectif f appartenant à l'ensemble de toutes les fonctions objectifs possibles et imaginables $\mathcal{F}$ qui peut être défini sur un domaine $\mathcal{X}$, l'ensemble image de ces fonctions est noté $\mathcal{Y}$. Après t itérations que l'algorithme a réalise, l'ensemble des points parcourus est noté d_t et vaut donc $\{(d_t^x(1), d_t^y(1)), \ldots, (d_t^x(t), d_t^y(t))\}$ où $d_k^x(k)$ et $d_k^y(k)$ représentent la $k^{\text{ième}}$ valeur parcourue dans l'ensemble $\mathcal{X}$ et, respectivement, dans $\mathcal{Y}$. En toute généralité, ils définissent la distribution probabiliste suivante pour n'importe quelle fonction $f \in \mathcal{F}$:

$$p(f) = p(f(x_1), \ldots, f(x_{|\mathcal{X}|})),$$

où $|\ \mathcal{X}\ |$ représente la cardinalité de l'ensemble $\mathcal{X}$, qui est toujours fini, comme ce théorème est applicable pour tous les problèmes d'optimisation. Par cette distribution, ils définissent la probabilité conditionnelle $p(d_t^y \mid f, t, a)$ qui donne la probabilité d'obtenir l'échantillon d_t^y après t itérations de l'algorithme a sur une fonction objectif $f \in \mathcal{F}$. De plus, par cette formule, il est possible de mesurer la performance de l'algorithme a et deux mêmes probabilités conditionnelles donnent la même performance. Leur théorème peut être cité à présent.

Théorème 1.5.1 ***(No Free Lunch Theorem)***
En fixant l'ensemble des points parcourus dans l'espace image d_t^y et le nombre d'itérations m, nous avons pour n'importe quelle paire d'algorithmes a_1 et a_2 :

$$\sum_{f \in \mathcal{F}} p(d_t^y \mid f, t, a_1) = \sum_{f \in \mathcal{F}} p(d_t^y \mid f, t, a_2).$$

En d'autres mots, la somme sur toutes les fonctions objectifs, possibles et imaginables, de la probabilité d'obtenir le même ensemble de trajectoires dans l'espace image avec deux algorithmes différents, est la même.

Ce théorème signifie donc que, sur toutes les fonctions objectifs possibles et imaginables, la performance de tous les algorithmes d'optimisation combinatoire se vaut.

1.6 Réglage des paramètres

Précédemment, dans la Figure 1.1, nous avons constaté que les trois opérations de l'AG comportent des paramètres et que différents opérateurs peuvent réaliser ces opérations. En accord avec le *No Free Lunch Theorem* abordé précédemment, il faut donc régler ceux-ci pour améliorer la rapidité et la précision de cet algorithme, pour chacun des problèmes d'optimisation que l'AG devra résoudre. Eiben et al. [6] classifient les réglages des paramètres en deux sous-catégories. La première comporte ceux réalisés avant l'exécution principale du programme sur une fonction donnée. C'est l'**ajustement des paramètres** (*Parameter Tuning*). La deuxième est composée des réglages qui se réalisent durant l'exécution du programme. C'est le **contrôle des paramètres** (*Parameter Control*). Ces contrôles peuvent être déterminés avant l'exécution ou s'adapter durant celle-ci, grâce à un feedback d'état (simple adaptation) ou en utilisant l'évolution (auto-adaptation). La Figure 1.4 synthétise ces différents termes.

Un contrôle déterminé est, par exemple, le fait de varier la valeur du paramètre après un certain nombre d'itérations sans regarder si c'était le moment adéquat pour le faire. Le contrôle adaptatif,

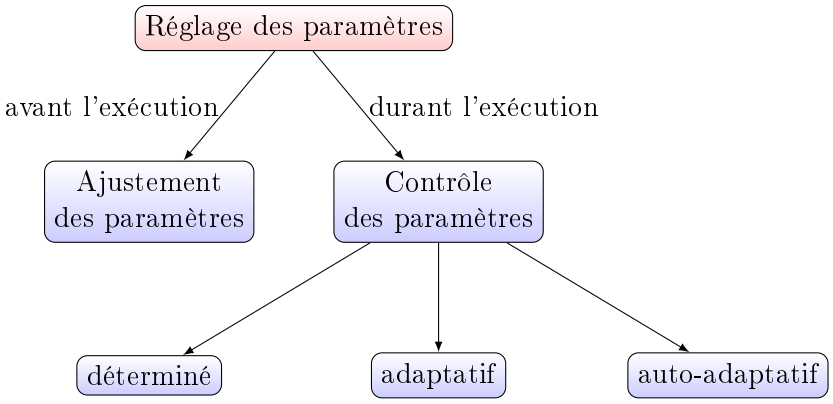


FIGURE 1.4 – Ce schéma représente la classification des différents types de réglages des paramètres. La même terminologie peut être appliquée pour ceux réalisés sur les opérateurs. Ce schéma est inspiré par celui présent dans l'article d'Eiben et al. [6].

quant à lui, observe l'évolution de la population et, par exemple, peut ainsi augmenter le taux de mutation si la population comporte que des individus fortement semblables. Le contrôle auto-adaptatif suggère d'utiliser l'évolution pour obtenir des valeurs pour les paramètres qui seront jugées idéales pour une population donnée et un problème donné.

Ces deux derniers contrôles permettent de ne plus régler les paramètres de l'algorithme génétique pour chaque problème, ceci étant une tâche fastidieuse et recommandée par le *No Free Lunch Theorem*. De plus, une partie de tels contrôles est réalisée en mettant dans les individus des valeurs pour les paramètres qui doivent être optimisés. Ainsi, Pellerin et al. [13] décomposent le génotype d'un individu en deux. Une partie, appelée **exon**, représente les variables du problème d'optimisation considéré. Donc, il y a autant d'exons que de dimensions au problème. Dans l'algorithme génétique, les exons sont également appelé **chromosomes**. L'autre partie représente les gènes ajoutés à l'individu mais qui n'ont aucune influence sur la fonction d'évaluation. Ceux-ci sont appelés **introns**.

En réalité, Eiben et al. [6] classifie les techniques pour régler les paramètres de différentes manières. Celle abordée dans les paragraphes précédents relève du *comment* : comment le réglage est réalisé. Trois autres manières y sont développées : le *quoi*, la *preuve* et la *portée*. Le *quoi* signifie ce qui change. En effet, ce mémoire se concentre sur les paramètres qui peuvent être réglés. Notons qu'il est également possible de régler les opérateurs (ce qui sera abordé brièvement en Section 2.1.6), la représentation d'un individu, la fonction d'évaluation et bien d'autres. La *preuve*, elle, aide le contrôle des paramètres. C'est-à-dire que lors de l'exécution, il y a un processus de surveillance qui observe si tel ou tel changement est bénéfique ou serait bénéfique. La *portée* représente le niveau de changement pour un paramètre contrôlé. Ce sont les éléments de l'algorithme qui sont touchés par un tel changement. Par exemple, le taux de mutation a forcément une influence directe sur la mutation même, mais également sur la diversification des individus, sur la convergence de l'algorithme et sur la précision de la réponse heuristique que celui-ci fournit. La portée est difficile à mesurer car les algorithmes génétiques sont des méthodes heuristiques. Bien entendu, dans le reste de ce mémoire, ce qui nous intéresse le plus c'est comment faire le changement et la preuve utilisée pour le réaliser. En ce qui concerne le *comment*, nous préférerons les contrôles adaptatifs et auto-adaptatifs car ceux-ci tiennent compte de l'état de l'exécution de l'algorithme.

L'étude réalisée par Karafotias et al. [14] et celle réalisée par Eiben et al. [6] répertorient différents réglages pour les paramètres et opérateurs de cet algorithme. Dans le prochain chapitre, une revue de ces possibilités sera développée afin de prendre connaissance des méthodes existantes.

Chapitre 2

Présentation des différents réglages existants pour les paramètres et les opérateurs d'un algorithme génétique

Dans ce chapitre, nous expliquerons différents réglages existants pour les paramètres et les opérateurs, sans être critique. La première section concernera les paramètres, ce qui est le centre de ce mémoire. Notons que nous citerons aussi quelques contrôles déterminés pour faire une revue complète, malgré que nous retiendrons que des contrôles adaptatifs et auto-adaptatifs. Cette revue provient principalement des articles de Karafotias et al. [14] et d'Eiben et al. [6]. Bien entendu, nous avons consulté les articles cités pour compléter leurs résultats. Ensuite pour être complet, dans la deuxième section, nous citerons quelques contrôles d'opérateurs, bien que nous n'en ferons pas dans ce mémoire. Ceci restera donc un domaine à approfondir. Et dans la troisième section, nous mentionnerons un article, dans lequel l'auteur a combiné des réglages auto-adaptatifs concernant les paramètres avec les opérateurs, afin de montrer que de tels réglages peuvent être réalisés.

2.1 Différents réglages existants pour les paramètres

Dans cette section, nous commencerons par expliquer les différents réglages concernant la taille de la population. Puis nous citerons des contrôles du paramètre η pour la mutation de Gauss. Nous continuerons par expliciter ceux qui règlent le taux de mutation p_m et le taux de croisement p_c. Ensuite, nous citerons un réglage valable pour n'importe quel paramètre. Pour finir, nous reprendrons des articles existants qui combinent différents réglages et montrent que cela fonctionne bien.

2.1.1 La taille de la population μ

Selon l'article de Karafotias et al. [14], la taille de la population peut être :
— approximée (en exécutant des populations en parallèle) ;
— contrôlée de manière déterminée ou adaptative ;
— enlevée, c'est-à-dire ne plus en tenir compte (ce qui revient à faire du contrôle auto-adaptatif).
Les méthodes qui suivent ont été répertoriées par Karafotias et al.

Approximer la taille

L'article d'Harik et Lobo [15] propose une méthode nommée *parameter-less GA*. Celle-ci consiste à faire évoluer plusieurs populations de tailles différentes en parallèle. En ordonnant ces populations par taille croissante, ils contraignent que la population suivante ait le double de la taille de la précédente. Ceci impose donc l'introduction d'un nouveau paramètre : la taille de la plus petite

population. Ils commencent avec un certain nombre de populations, ceci représente un second paramètre. Puis, quand une population devient plus performante qu'une plus petite, cette dernière est supprimée. La performance d'une population est calculée par sa valeur de fitness moyenne, dans leur article. Cependant, aucune population de taille différente n'est recréée durant l'exécution. Ainsi, la meilleure population reste et continue à être modifiée durant plusieurs autres générations jusqu'à l'obtention d'une solution heuristique. Ils appuient leur méthode par l'argument suivant : comme une petite population peut-être évaluée plus souvent qu'une grande, la valeur de fitness moyenne de la plus petite devrait être meilleure que celle de la plus grande. Ainsi, ils espèrent qu'une population de petite taille survive à cette sélection.

Dans l'article de Smorodkina et Tauritz [16], seules deux populations sont exécutées en parallèle tout le temps. L'une est composée du double d'individus par rapport à l'autre. La plus petite (notée P_t à l'itération t) est supprimée si la plus grande (P_{t+1}) devient plus performante. De même, la performance d'une population est mesurée grâce à la valeur moyenne de fitness, dans leur article. Si la moyenne de fitness de P_{t+1} dépasse de 5% celle de P_t, alors P_t est supprimée. Ou si la valeur de fitness maximale de P_t manque d'amélioration après 800 itérations et si la valeur moyenne de P_{t+1} est plus grande que celle de P_t, alors P_t est supprimée. Notons que ces deux paramètres (5% et 800) ont été fixés dans leur étude sans analyse. Si un nombre maximal d'évaluations de fonctions est imposé, alors il faut ajouter au sein de cette dernière condition l'exigence suivante : le nombre d'évaluations non utilisées par P_t doit être au moins aussi grand que le nombre d'évaluations utilisées par P_{t+1} pour que la population P_t soit supprimée. Après la suppression de celle-ci, une population P_{t+2} est créée et comporte le double d'individus par rapport à la population P_{t+1}, puis t est incrémenté d'une unité. Cette méthode, nommée *GPS-EA*, serait deux fois plus rapide qu'une population de taille optimale fixée, selon Karafotias et al. [14]. Cependant, plusieurs paramètres ont été ajoutés comme le taux de dépassement (de 5%) et le nombre d'itérations sans amélioration notable. De plus, le choix de la moyenne de fitness impose que tous les individus de la population soient bons, or normalement il est préférable d'avoir quelques individus moins bons. En effet, cela permet de parcourir l'espace de faisabilité pour se rassurer de ne pas avoir omis d'autres minima. La médiane pourrait alors être préférée.

Une méthode semblable est présentée dans l'article d'Hinterding et al. [17]. Trois populations sont exécutées en parallèle, sans être comparées durant un certain nombre d'itérations. Après chaque période, une comparaison est réalisée pour prendre des décisions d'expansion, de compression ou d'échange concernant la taille des populations. Cette méthode nommée *IGA* comporte donc un nouveau paramètre : la période.

Contrôler la taille

Un contrôle déterminé est proposé par Koumousis et Katsaras [1]. Nous la développerons dans Minamo dans le Chapitre 4. Ils proposent de commencer l'algorithme avec une population de taille maximale. La taille de la population est réduite linéairement durant plusieurs itérations. Puis périodiquement, la population est restaurée à sa taille maximale. Cette diminution linéaire en partant d'une taille maximale représente une dent, si nous considérons l'axe des abscisses comme le numéro de l'itération et l'axe des ordonnées comme la taille de la population. La combinaison de diminutions et d'augmentations représente une scie. Cette méthode est appelée *Saw-Tooth*.

Des contrôles adaptatifs existent. Par exemple, Smith et Smuda [18] proposent une estimation continue de la perte de fitness due à l'erreur de la sélection. Cette estimation est ensuite comparée à la perte de fitness réelle. Cependant, ceci semble fort approximatif. Sinon, Yu et al. [19] estiment la taille de la population pour la génération suivante selon des *Building Blocks*. Cette méthode est nommée *DSMGA*. Mais cela demeure encore approximatif.

Enlever la taille (ne plus en tenir compte)

Pour enlever la taille de la population, Eiben et al. [20] et Teo [21] proposent de mettre dans le génotype de chaque individu une valeur comprise entre deux bornes qui permet de déterminer la taille de la prochaine génération. Pour se faire, Eiben et ses collègues réalisent une somme de ces valeurs tandis que Teo opère une moyenne. Cependant, deux nouveaux paramètres ont été introduits.

Notons que cette valeur intégrée pour chaque individu est mutée durant les générations. Ainsi, Eiben et al. [20] cite l'article de Back et Schutz [22] qui propose une mutation pour n'importe quel nombre p compris entre deux bornes inscrit dans les introns. Quand p doit être muté, un nombre q est tiré aléatoirement suivant une loi normale de moyenne valant zéro et de variance valant un. Et la probabilité devient :

$$p' = \left(1 + \frac{1-p}{p} e^{-\gamma q}\right)^{-1}$$

où γ est un taux d'apprentissage. Ainsi, pour une valeur p comprise entre deux bornes, il suffit de réaliser un shift sur celle-ci pour la ramener entre zéro et un, puis d'appliquer le shift inverse sur p' obtenu.

Une autre philosophie est proposée par Arabas et al. [2]. Leur méthode s'appelle *GAVaPS* pour l'abréviation en anglais *Genetic Algorithm with Varying Population Size*. Chaque individu i à sa création se voit attribuer dans son génome (en *introns*) une espérance de vie $EV(i)$ qui dépend de sa valeur de fitness $fit(i)$, de la valeur moyenne de la population fit_{moy}, de celle du meilleur individu fit_{max} et du pire fit_{min}. Cette espérance de vie est tout de même bornée par deux paramètres : l'espérance minimale EV_{min} et maximale EV_{max}. Après chaque itération, chaque individu est vieilli et si son âge, qui est aussi codé dans son génome, dépasse son espérance de vie alors celui-ci est retiré de la population et ne fera pas partie de la génération suivante. Bäck et al. [23] proposent de ne pas incrémenter l'âge du meilleur individu, ce qui représente une forme d'élitisme. L'espérance de vie est donc obtenue comme suit

$$EV(i) = \begin{cases} EV_{min} + \eta \dfrac{fit(i) - fit_{min}}{fit_{moy} - fit_{min}} & \text{si } fit_{moy} \geq fit(i) \\ \dfrac{1}{2}(EV_{min} + EV_{max}) + \eta \dfrac{fit(i) - fit_{moy}}{fit_{max} - fit_{moy}} & \text{si } fit_{moy} < fit(i) \end{cases}$$

où $\eta = \dfrac{1}{2}(EV_{max} - EV_{min})$.

Nous coderons cette méthode au sein de Minamo (voir Chapitre 5).

Pour ne pas ajouter de paramètres, Cook et Tauritz [3] définissent la méthode *FiScIs-EA*. Cette abréviation signifie en anglais *Fitness Scaled Individual Survival Evolutionary Algorithm*. Celle-ci consiste à déterminer une probabilité de survie pour chaque individu i, notée $p_{survie}(i)$, dépendant des valeurs de fitness de l'individu, du meilleur et du pire. Si un nombre tiré aléatoirement est plus petit que p_{survie}, alors l'individu reste pour la génération suivante, sinon il est supprimé. Ainsi, même les moins bons ont quand même une chance de survivre. Ce taux est déterminé par la mise à l'échelle linéaire suivante :

$$p_{survie}(i) = \frac{fit(i) - fit_{min}}{fit_{max} - fit_{min}}.$$

Cette méthode sera la dernière implémentée dans Minamo (voir Chapitre 6).

Cependant, pour être complet et réaliser un état de l'art étoffé, nous continuerons notre analyse dans les sections suivantes pour les différents paramètres. En réalité, nous nous concentrerons uniquement sur le réglage de la taille de la population.

2.1.2 La sélection

Les méthodes qui suivent ont été référencées par Karafotias et al. [14]. Les deux premiers réglages concernent la sélection par tournoi réalisée pour générer la population suivante. Les deux suivants règlent la sélection qui précède le croisement.

La première méthode que nous citons concilie le recuit-simulé et la sélection par tournoi. Elle est introduite par Goldberg [24] et est nommée en anglais *Boltzmann Tournament Selection*. Elle est réalisée lors de la création de la génération suivante. Il faut donc piocher des individus au sein de la génération actuelle et ses enfants. Dans son article, cette opération est réalisée en cinq étapes. D'abord, il faut choisir un premier individu aléatoirement. Après cela, un second individu est tiré aléatoirement. Cependant, il doit être différent du premier. Ensuite, un troisième individu est sélectionné. Il doit être (strictement) différent des deux autres individus. Puis, une première compétition (d'anti-acceptation) est réalisée entre le deuxième et le troisième individu. Enfin, la seconde compétition est appliquée entre le meilleur individu gardé de la première compétition et le premier individu tiré. Ces étapes sont répétées tant que la nouvelle population n'est pas complète.

Affenzeller [25] propose deux contrôles adaptatifs utilisant la ségrégation d'algorithmes génétiques, c'est-à-dire qu'il utilise le croisement et la mutation de l'algorithme génétique, le mécanisme de température qui provient du recuit-simulé et qui permet de varier la pression de sélection. Cela permet d'éviter une convergence prématurée. L'idée de la première méthode est de sélectionner des individus parmi les enfants générés par croisement et mutation. Le nombre de ces enfants étant déterminé en multipliant la taille de la population, fixe, par le facteur $Temp$, représentant la température et devant être plus grand ou égal à un. La seconde méthode, quant à elle, sélectionne parmi la génération courante et ses enfants. Le nombre d'enfants est déterminé de la même manière que la première méthode.

Comparé aux deux réglages précédents, qui concernaient la sélection des individus pour la génération suivante, la méthode proposée par Kaveh et Shahrouzi [26] adapte la sélection des individus pour le croisement. Elle utilise également la ségrégation d'algorithmes génétiques, en utilisant la mutation issue de l'algorithme génétique et une population séparée de la population courante provenant de l'algorithme des colonies de fourmis. L'idée est de maintenir deux populations, la courante et la colonie, afin de sélectionner des parents parmi la population courante et la colonie. Cette dernière comporte les bons individus de générations précédentes afin de garder de bons gènes et est de taille variable, tandis que la population garde une taille fixe. Cette méthode sélectionne le même nombre d'individus que la taille de la population.

Eiben et al. [20] encodent la taille des tournois réalisés avant le croisement dans le génome des individus. Sa méthode est nommée *GASAT*, pour *Genetic Algorithm Self-Adaptive Tournament Size Parameter*. Cette valeur est comprise entre deux bornes et la somme de toutes les valeurs individuelles représente la taille du tournoi. Ces valeurs sont mutées selon le même schéma spécifique décrit pour la mutation du paramètre concernant sa proposition pour la taille de la population (se référer à la Section 2.1.1).

2.1.3 La mutation de Gauss

Comme décrit dans la Section 1.4, la mutation de Gauss comporte un paramètre : la taille d'un pas η. Nous allons donc citer des méthodes permettant de contrôler celle-ci.

Eiben et al. [6] suggèrent que η doit être proportionnel à la distance entre la solution globale et l'individu qui doit être muté. Il appuie son idée avec l'argument suivant. Loin de la solution (avec une distance initiale d_0 élevée) et avec un η fixé, un individu qui n'est que muté prendra en moyenne

$\frac{d_0}{\eta}$ itérations pour atteindre l'optimum. Mais en arrivant près de la solution, la précision pour atteindre cet optimum dépendra de η. Ainsi η doit être proportionnel à la distance de l'optimum à chaque itération. Cependant, en pratique, l'implémenter est presque irréalisable car, pour ce faire, il faudrait connaître l'optimum mais c'est ce que l'AG cherche.

C'est pourquoi, Eiben et al. [6] suggèrent d'utiliser la règle du 1/5 de Rechenberg qui est un contrôle adaptatif. Après chaque création de générations, il faut vérifier si, sur l'entièreté des individus mutés, un cinquième de ceux-ci possèdent une fonction de fitness plus élevée qu'auparavant. Dans ce cas, cela signifie que les points sont loin de l'optimum et que η peut être augmenté, en étant multiplié par deux, par exemple. Dans le cas contraire, il faut le diminuer en le divisant par deux, afin de s'approcher de l'optimum. Malheureusement, Eiben dit que cette règle induit trop rapidement en erreur l'algorithme qui reste bloqué près de minima locaux.

Un contrôle auto-adaptatif est proposé dans l'article d'Hinterding et al. [17] qui est cité dans l'article de Karafotias et al. [14]. Il propose de coder dans chaque individu une valeur η aléatoire comprise entre 0.000001 et 0.2. Puis lors de la mutation d'un individu, les étapes sont les suivantes :

1. décoder son η ;
2. appliquer la mutation de Gauss sur ce η avec une déviation standard de 0.013 pour obtenir $\tilde{\eta}$;
3. pour les autres gènes de cet individu, utiliser $\tilde{\eta}$ comme nouvelle taille du pas ;
4. dans les introns de ce nouvel individu, écrire $\tilde{\eta}$.

L'avantage de cette mutation est que l'algorithme détermine les valeurs de η par lui-même grâce à l'évolution.

2.1.4 La probabilité de mutation p_m et la probabilité de croisement p_c

L'article d'Eiben et al. [6] cite celui d'Hesser et Männer [27] qui propose un contrôle déterminé pour la probabilité de mutation p_m. Cette probabilité est calculée mathématiquement en introduisant de nouvelles constantes (θ, β et ν) représentant des coefficients de proportionnalité pour un temps d'attente et un temps additionnel. Le temps d'attente t_w est dû à un phénomène d'absorption causé par le croisement, tandis que le temps additionnel t_l est induit par la mutation. Ceux-ci peuvent être décrits par un système d'équations, qui en le résolvant nous permet de déterminer la probabilité de mutation pour chaque itération t, comme suit :

$$p_m(t) = \sqrt{\frac{\theta}{\beta}} \frac{e^{\frac{-\nu t}{2}}}{\mu\sqrt{n_D}}.$$

Quant à lui, l'article de Karafotias et al. [14] présente sommairement un contrôle auto-adaptatif que réalise Bäck et al. [23] pour cette même probabilité. Formellement, Bäck qui a codé l'AG par des bits propose d'encoder sur dix bits une probabilité de mutation (comprise entre 0.001 et 0.25) au sein de chaque individu. Cette valeur est attribuée à chacun de manière aléatoire lors de l'initialisation de la population. Lors de la mutation d'un individu, p_m doit d'abord être décodé afin d'ensuite muter ces dix bits. Puis, après avoir décodé ces nouveaux bits et obtenu la nouvelle valeur $\tilde{p_m}$, les gènes restants de l'individu sont mutés grâce à ce taux. Eiben et al. [6] dit que les individus sont coincés près d'optimums locaux avec $p_m \approx 0$ et que la méthode n'est valable que pour le codage en bits et non en réel. Cependant, il serait facile de l'adapter au cas réel en considérant chacun des gènes à muter comme des $x \in \mathbb{R}^1$ dans la formule de la Section 1.4.

Pour la probabilité de croisement, Bäck et al. [23] propose également un contrôle auto-adaptatif. A nouveau, dans chaque individu codé par des bits, ce taux est encodé sur dix bits et est compris entre zéro et un. Cette méthode utilise la sélection par tournoi, et chaque croisement est réalisé

entre deux parents. Pour un couple de parents, un nombre aléatoire est tiré selon une loi uniforme $[0, 1]$. Selon ce nombre, trois possibilités se présentent. Premièrement, si celui-ci est plus petit que les taux encodés dans les deux parents, alors ceux-ci sont croisés uniformément avec une probabilité 0.5 pour donner naissance à deux enfants, qui sont alors directement injectés dans la population après avoir été mutés. Deuxièmement, si ce nombre est plus grand que ces deux probabilités, alors les parents ne sont pas croisés mais ils sont mutés pour être réinjectés dans la population immédiatement. La troisième possibilité survient quand le nombre aléatoire est compris entre les deux valeurs des individus. Dans ce cas, celui ayant son taux plus petit que le nombre tiré est muté et injecté immédiatement dans la population. L'autre individu est maintenu, puis une sélection par tournoi est organisé pour obtenir un nouvel individu. Ensuite, l'opération est répétée sur ces deux membres : l'individu restant et l'individu nouvellement tiré.

Pour ces deux probabilités, Srinivas et Patnaik [28] proposent un contrôle adaptatif qui tient compte de la convergence. Celle-ci est détectée lorsque la moyenne des fitness fit_{moy} approche la meilleure valeur de fitness de la population fit_{max}. Le taux de convergence est alors $fit_{max} - fit_{moy}$. Le but est de maintenir la diversité en augmentant la variation lorsque c'est nécessaire et de diminuer les variations quand la population est déjà fortement diversifiée. Ainsi, le taux p_c est défini par une fonction linéaire qui est proportionnelle à la différence entre la plus grande valeur de fitness des parents fit' et celle du meilleur individu. Cette fonction doit également être inversement proportionnelle au taux de convergence. Pour le taux de mutation, un raisonnement similaire peut être fait. Ils introduisent naturellement les deux calculs suivants en n'oubliant pas que ces probabilités appartiennent à l'intervalle $[0, 1]$

$$p_c = \begin{cases} k_1 \dfrac{fit_{max} - fit'}{fit_{max} - fit_{moy}} & \text{si } fit' \geq fit_{moy} \\ k_3 & \text{sinon} \end{cases}$$

$$\text{et} \quad p_m = \begin{cases} k_2 \dfrac{fit_{max} - fit}{fit_{max} - fit_{moy}} & \text{si } fit \geq fit_{moy} \\ k_4 & \text{sinon} \end{cases}$$

où fit est la valeur de fitness de l'individu qui doit être muté, et où k_1, k_2, k_3 et k_4 sont de nouvelles constantes. Dans leur étude, ils ont assigné à k_2 et k_4 la valeur 0.5 et pour k_1 et k_3 la valeur 1. Ils ont alors prouvé que leur méthode, qui combine ces deux réglages, était plus performante que l'AG sans réglages. Ce qui nous mène à considérer la combinaison de tels réglages dans une section suivante, mais avant citons un réglage qui est valable pour n'importe quel paramètre.

2.1.5 Un réglage supplémentaire valable pour n'importe quel paramètre

Karafotias et al. [14] cite l'article de Wong et al. [29] qui suggère de diviser l'exécution en paires de périodes. Le nombre d'itérations constituant une paire représente donc un nouveau paramètre. Pour chaque paire, la première sous-période débute par un choix aléatoire de trois valeurs dans l'ensemble des valeurs que peuvent prendre le paramètre qu'il faut régler. Après cela, l'exécution prend aléatoirement une valeur parmi celles-ci, à chaque fois que l'opérateur concerné a besoin de son paramètre. Puis, après l'exécution de cette première sous-période, l'algorithme calcule une somme des scores obtenus pour chacune de ces trois valeurs, puis les pondère pour obtenir des probabilités. Ainsi, durant la deuxième sous-période, à chaque fois que le paramètre doit être utilisé, l'exécution prend une des trois valeurs selon la probabilité calculée, en tirant un nombre aléatoire. Lorsque la paire de période est terminée, il faut recommencer ces différentes étapes. Notons qu'Aleti et Moser [30] proposent de créer un sous ensemble de valeurs pour le paramètre s'il est réel.

2.1.6 La combinaison de réglages concernant les paramètres

Bäck et al. [23] démontrent, par trente exécutions partant de la même population sur cinq fonctions tests, que combiner leurs trois méthodes auto-adaptatives concernant la taille de la population

(décrite en Section 2.1.1), la probabilité de croisement et la probabilité de mutation (tous deux décrits en Section 2.1.4), est plus avantageux que la simple version de l'AG ou que leur utilisation séparée. Cette méthode combinée est appelée *SAMXPGA*.

Hinterding et al. [17] combinent leur auto-adaptation de η (Section 2.1.3) avec leur contrôle adaptatif de la population (Section 2.1.1). Par dix exécutions consécutives sur dix fonctions tests, ils montrent que cela est bénéfique.

Maruo et al. [31] ainsi qu'Eiben et al. [20] mettent tout dans le génome, à savoir η, p_m, p_c et la taille de la sélection par tournoi. Cette dernière est déterminée par la majorité des valeurs (pour Maruo et al.) et par une somme de ces valeurs (pour Eiben et al.). Kramer [32] et Meyer-Nieber avec Beyer [33] ont démontré la puissance de cette méthode, en particulier pour les espaces de recherche codés par des réels.

L'article de Mc Ginley et al. [34] combine des contrôles adaptatifs pour la mutation, le croisement et la sélection. Deux populations distinctes sont utilisées : une permettant l'exploration et l'autre l'exploitation. Le meilleur individu d'une génération est gardé pour la suivante. Pour l'exploitation, pour chaque paire d'individus sélectionnés par tournoi, ils sont croisés uniformément puis leur enfant subit une faible mutation $p_m = 0.01$. Pour l'exploration, chaque individu sélectionné aussi par tournoi est muté en utilisant la probabilité p_m calculée par une méthode qu'ils ont introduite dans leur article et qui est nommée *SPD*. Les sélections par tournoi sont déterminées par une méthode nommée *HPD* qu'ils ont également développée dans leur article. Notons que, dans cette méthode, la probabilité de croisement p_c est également déterminée par la méthode *SPD* et signifie le choix de l'opérateur de croisement pour générer un enfant : si un nombre aléatoire est plus petit que p_c alors la méthode de l'exploitation sera utilisée sinon ce sera celle de l'exploration.

Dans le dernier paragraphe, nous constatons que les auteurs ont réalisé une combinaison d'opérateurs pour contrôler les paramètres de manière adaptative. Dans la section suivante, nous citerons quelques réglages intéressants qui concernent le choix des opérateurs.

2.2 Différents réglages existants pour les opérateurs

Dans cette section, nous commencerons par énoncer un réglage qui peut être réalisé sur la sélection. Puis, une méthode concernant le croisement sera développée. Nous terminerons par une combinaison de réglages des opérateurs. Les deux premiers réglages sont en réalité des contrôles auto-adaptatifs, tandis que le troisième est adaptatif.

Répertoriée par Karafotias et al. [14], la méthode proposée par Smorodkina et Tauritz [35] concerne le réglage de l'opérateur de sélection de parents. Dans la Section 1.4, nous avons vu qu'il existait plusieurs opérateurs de sélection de parents. Ainsi, choisir quel opérateur utiliser est une tâche fastidieuse. C'est pourquoi ils proposent de mettre dans le génotype de tous les individus une préférence d'opérateur de sélection, grâce à une représentation pour la fonction de sélection. Ainsi, un individu est sélectionné d'une certaine manière qui reste fixe pour toute l'exécution de l'AG. Puis celui-ci indique au programme de quelle manière il désire voir ses compagnons amoureux être sélectionnés. Par exemple, un premier individu est sélectionné aléatoirement. Dans celui-ci, il est noté qu'il désire la sélection par un tournoi de Boltzmann entre trois individus pour sélectionner son partenaire. Ainsi, trois autres individus sont sélectionnés aléatoirement afin d'obtenir un seul partenaire, selon le schéma décrit en Section 2.1.2. Dans leur papier, ils imposent que l'accouplement se fasse toujours entre deux individus.

L'article d'Eiben et al. [6] explique de manière détaillée le contrôle de Davis [36] concernant le réglage d'opérateurs pour le croisement. Différents opérateurs op_k, ayant chacun leur propre taux de croisement $p_c(op_k)$, sont utilisés. De plus, chaque opérateur a sa propre valeur "delta locale" δ_k, qui représente la force de l'opérateur op_k et est mesurée par l'avantage qu'un enfant a par rapport à ses parents. Après K itérations, il faut recalculer $p_c(op_k)$ par la formule suivante :

$$p_c(op_k) \leftarrow 0.85\ p_c(op_k) + \delta_k^{norm},$$

où $\sum_k \delta_k^{norm} = 0.15$.

Karafotias et al. [14] répertorient la méthode d'Herrera et Lozano [37] qui combine deux opérateurs de croisement, un ayant des propriétés d'exploitation et l'autre d'exploration, avec une probabilité p_e qui permet de choisir lequel appliquer. Dans la Section 4.1 de leur article, ils citent plusieurs mesures de diversité concernant le génotype, dont la distance euclidienne, et, dans la Section 4.2, ils énumèrent trois mesures concernant la diversité du phénotype, dont deux provenant de Lee et Takagi [38]. De plus, une sélection par classement linéaire est utilisée et est dirigée par un paramètre SP_{min}, représentant la pression de sélection minimale. Pour contrôler ces deux paramètres, ils décident d'utiliser les *Fuzzy Logic Controllers* (*FLC*). Leur construction utilise la distance euclidienne et la première mesure de Lee et Takagi : selon les différents degrés de ces mesures (faible, moyen et fort), l'ajustement des deux paramètres varie. Herrera et Lozano [37] ont démontré par quatre fonctions tests, qu'utiliser leur méthode nommée *ARGAF* pour *Adaptive Real-Coded GA Based on FLC* est plus bénéfique qu'utiliser un simple AG utilisant un croisement qui associe des individus qui ont un niveau de diversité élevé, comme cela se fait naïvement bien souvent.

Bien entendu, d'autres réglages existent et certains sont énoncés en page 9-10 de l'article de Karafotias et al. [14].

2.3 Combinaison de réglages pour les paramètres et les opérateurs

L'article de Pellerin et al. [13] nous donne l'exemple qui combine des réglages sur le taux de mutation p_m et la probabilité de croisement p_c avec des réglages sur les opérateurs de croisement et de mutation. Pour ce faire, ils utilisent cinq opérateurs de croisement et cinq opérateurs de mutation. De plus, dans chaque individu, un gène indique un opérateur de croisement et un second l'opérateur de mutation. En plus de ceux-ci, deux gènes supplémentaires encodent des valeurs pour p_c et p_m. Lors de la création de la première génération, ces probabilités sont uniformément distribuées entre zéro et un. De même, les opérateurs sont distribués aléatoirement entre ces nouveaux individus. Le croisement est toujours réalisé entre deux individus et utilise les paramètres du meilleur, c'est-à-dire celui qui possède une valeur de fitness plus élevée. La mutation est déterminée selon le meilleur individu de la population. Le problème du voyageur de commerce est utilisé pour comparer leur méthode avec un algorithme génétique basique, utilisant un croisement à deux points, une mutation à une position aléatoire, un taux de croisement de 0.75 et un taux de mutation de 0.01. Ils concluent que leur méthode permet de trouver la distance minimale en moins de génération et une distance plus petite que celle issue de l'AG sans adaptation. De plus, ils remarquent que, selon la population initiale et le problème, leur méthode choisit les paramètres adéquats.

2.4 Résumé des réglages importants concernant le paramètre μ

Nous constatons que les réglages de paramètres introduisent bien souvent des nouveaux paramètres. Seuls certains, en particulier les auto-adaptatifs, le font rarement. Ainsi, nous savons que déterminer une méthode pour remplacer un paramètre va nécessiter de déterminer les valeurs des paramètres introduits au sein de l'AG existant de Minamo.

Au sein de Minamo, nous allons développer uniquement des méthodes réglant la taille de la population comme ces méthodes vont pouvoir réduire la population pour converger vers un minimum et augmenter à nouveau le nombre d'individus pour introduire de la variabilité. Ainsi, nous retenons :

1. le contrôle déterminé de Koumousis et Katsaras [1], *Saw-Tooth*, qui fait décroître linéairement la population avant de l'augmenter à sa taille originale (voir Chapitre 4) ;
2. le contrôle adaptatif, *GAVaPS*, qui est introduit par Arabas et al. [2] et qui concerne l'ajout d'une espérance de vie à chaque individu (voir Chapitre 5) ;
3. le contrôle adaptatif nommé *FiScIs-EA* et introduit par Cook et Tauritz [3] qui calcule à chaque itération une probabilité de survie selon les valeurs de fitness maximale et minimale (voir Chapitre 6).

Chapitre 3

Présentation de Minamo et des progrès désirés

Minamo est un programme informatique développé par Cenaero, qui est un centre de recherche situé dans l'aéropôle de Gosselies, non loin de Charleroi. Ce programme résout des problèmes industriels de minimisation et est distribué à certains clients, dans différents domaines scientifiques. Initialement, cette entreprise était un centre de recherche en aéronautique, ce qui lui a donné son acronyme.

Minamo est une boîte à outils, comprenant différentes méthodes pour résoudre des problèmes de minimisation. Dans ce mémoire, nous n'utiliserons qu'une méthode globale dans laquelle nous avons intégré quelques stratégies pour tenter d'améliorer les résultats et la vitesse de convergence vers le minimum global. Comme nous l'avons déjà abordé, les problèmes que traite ce programme sont ceux développés pour l'industrie. Puisque les fonctions à évaluer nécessitent beaucoup de temps de calcul et de mémoire, il est peu concevable de pouvoir réaliser une minimisation directement sur celles-ci. En effet, ceci alourdirait considérablement l'exécution. Pour remédier à cela, les concepteurs de Minamo ont développé deux boucles imbriquées. La première permet de construire un méta-modèle. Celui-ci est une combinaison de fonctions élémentaires. Il a pour but d'approximer au mieux la fonction non-connue qui est à évaluer. Ceci sera développé dans la section suivante. La seconde boucle est l'algorithme génétique à proprement parler que nous avons décrit de manière générale dans la Section 1.4 et dont nous décrirons les valeurs de ses paramètres dans ce chapitre.

Notons que le présent travail a bénéficié de moyens de calcul mis à disposition sur le supercalculateur Tier-1 de la Fédération Wallonie-Bruxelles, infrastructure financée par la Région wallonne sous la convention n°1117545.

3.1 Optimisation assistée par modèles de substitution (*SBO*)

Cette section reprend des informations d'un document interne à Cenaero [39]. Nous expliquerons la boucle externe de Minamo, qui, d'abord, crée un méta-modèle, qui se rapproche au mieux du problème industriel à minimiser, ensuite l'optimise et le fait évoluer au fil des itérations. Cette méthode est appelée en français **optimisation assistée par modèles de substitution** et en anglais *Surrogate-Based Optimization*. Elle sera notée *SBO* dans la suite. Ce type d'optimisation est nécessaire dans les domaines industriels, car les fonctions à optimiser nécessitent un temps de calcul considérable lors d'une seule évaluation d'un point. Ainsi, pour éviter d'appeler abusivement ces fonctions coûteuses, il est nécessaire de construire un méta-modèle, un modèle de substitution, qui approchera au fur et à mesure la véritable fonction et dont l'évaluation en plusieurs points est moins coûteuse. Concrètement, dans une application industrielle, il faut d'abord définir le problème à résoudre ainsi que ses différentes contraintes. Ensuite, il faut concevoir l'expérience en initiali-

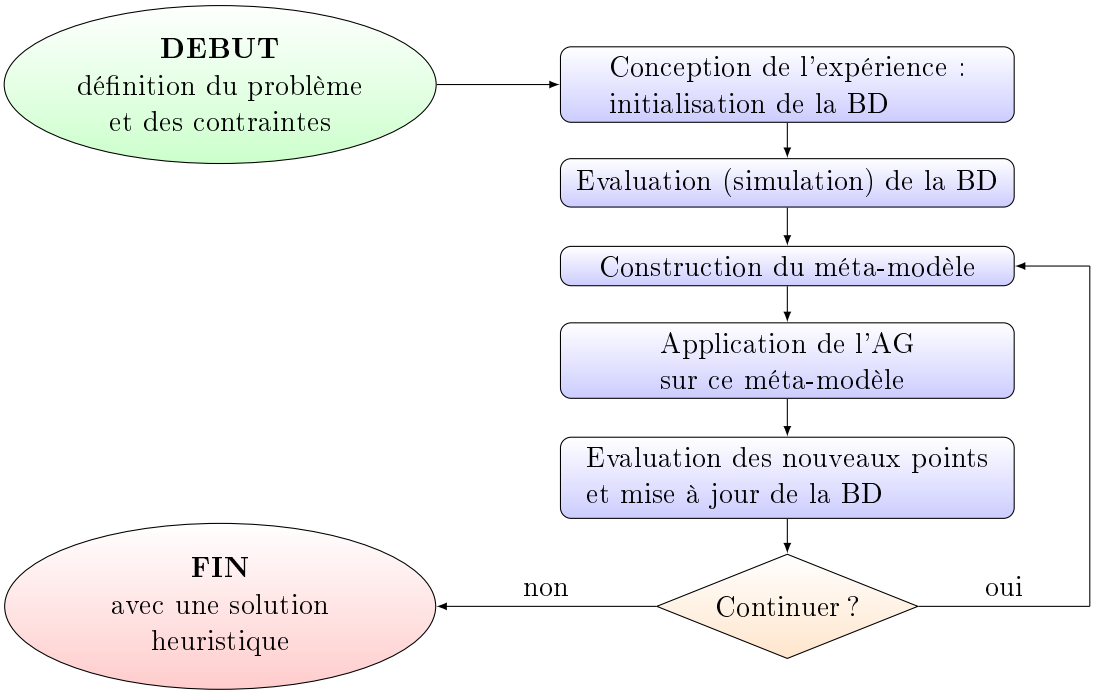


FIGURE 3.1 – Ce schéma représente l'optimisation assistée par modèles de substitution que Minamo utilise et est inspiré d'un document interne à Cenaero [39].

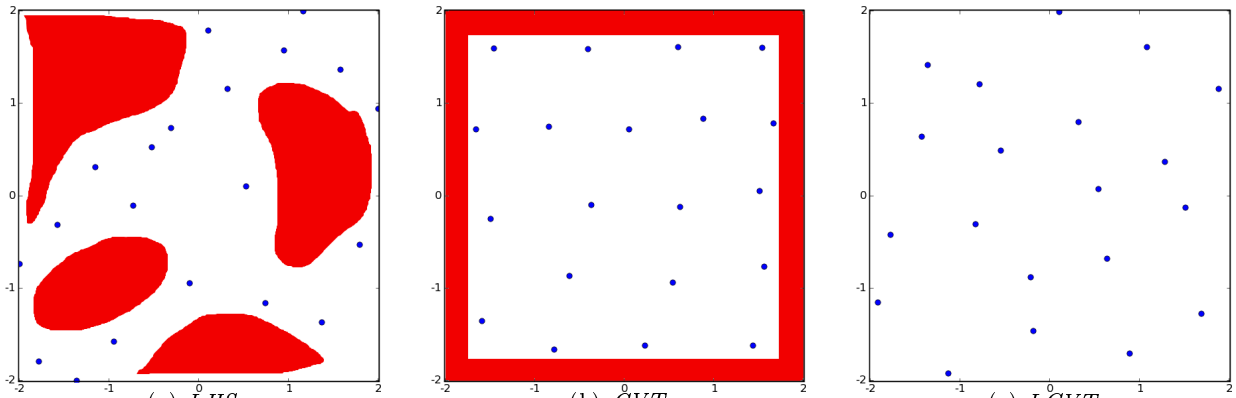
sant la base de données (BD) de cette boucle. Celle-ci est constituée principalement de $\overline{n}$ points dans l'espace de conception. Ces points sont générés par une certaine méthode (nous en citerons trois dans le paragraphe suivant). Bien entendu, ces points doivent alors être évalués par la véritable fonction coûteuse. Puis, la construction d'un méta-modèle est une nouvelle étape. Il existe différents algorithmes qui permettent de réaliser cette tâche (nous en verrons dans un paragraphe suivant). Ensuite, un algorithme heuristique permet de minimiser ce méta-modèle. Ici, nous utiliserons donc un algorithme génétique. Après cela, il faut incorporer au moins un nouveau point dans la BD, en exigeant une certaine distance entre les points déjà présents. Et pour finir, il faut se demander s'il est nécessaire de continuer ou non. Dans le cas affirmatif, l'algorithme recommence à l'étape de construction d'un méta-modèle. L'algorithme se termine avec une solution heuristique. La Figure 3.1 synthétise graphiquement ces étapes.

Citons à présent trois méthodes présentes dans Minamo qui permettent d'échantillonner l'espace de conception au début de cette boucle [40] ou au sein de l'AG :

— l'échantillonnage par hypercubes latins (*Latin Hypercube Sampling (LHS)*) ;
— la tesselation de centroïdes de Voronoï (*Centroidal Voronoi Tessellation (CVT)*) ;
— la latinisation de la tesselation centroïde de Voronoï ($LCVT$).

Dans la Figure 3.2, nous avons représenté l'échantillonnage en vingt points d'un espace en 2D par ces différentes méthodes. Les zones coloriées en rouge représentent les zones non couvertes par ces méthodes. Pour la méthode *LHS*, l'échantillonnage se fait en découpant l'espace de conception en différentes zones aléatoires (égale au nombre de points qui doivent être introduits). Puis un point est tiré au sein de chacune d'elles. Malheureusement, des zones peuvent restées non-couvertes lorsque peu de points sont générés, à cause de ce processus aléatoire. Par contre, la méthode CVT est un processus qui distribue uniformément les points dans l'espace. Cependant, aucun point n'est tiré au bord, car ce processus prend les centroïdes des régions. Pour finir, la stratégie $LCVT$ permet une meilleure distribution des points, en alliant les deux méthodes précédentes. Notons que l'avantage de *LHS* est qu'elle est plus rapide, qu'elle ne tient pas compte d'une région assez représentée (si

nous complétons une population) et que ses inconvénients s'atténuent avec une base de données contenant beaucoup de points (ce qui est le cas pour l'AG).



(a) *LHS* (b) *CVT* (c) *LCVT*

FIGURE 3.2 – La figure de gauche représente une BD de 20 points (en 2D) générée par *LHS*, celle du centre concerne une de *CVT* et celle de droite de *LCVT*. Les zones mises en rouge sont des zones non-couvertes par l'échantillonnage.

Pour créer un méta-modèle, y, défini pour tous les points de l'espace de conception, à partir d'un ensemble de points évalués par la véritable fonction objectif et ses contraintes, il existe différentes méthodes. Citons par exemple :

— les fonctions à base radiale (*Radial Basis Function (RBF)*) ;
— la régression par un processus gaussien (*Gaussian Process Regression* ou *Kriging*) ;
— les machines à vecteurs de support (*Support Vector Machines (SVM)*).

Les deux premières approches sont des méthodes d'interpolation tandis que la dernière est une méthode de classification. Les méta-modèles *RBF* [41] utilisés dans Minamo sont définis comme une combinaison linaire de $\overline{n}$ fonctions à base radiale h, où $\overline{n}$ est le nombre de points dans la BD. Ceci est traduit dans l'équation suivante :

$$y(x) = \sum_{k=1}^{\overline{n}} w_k \;\; h(\| x - \overline{x_k} \|_2, \; \sigma_k),$$

où w_k est le poids associé à la fonction à base radiale associée au point $\overline{x_k}$. Ces fonctions à base radiale, qui sont notées de manière générale $h(r, \sigma)$, ont comme paramètre la base radiale r et un méta-paramètre σ. Elles sont les gaussiennes (exprimées $\exp\left\{\frac{-r^2}{2\sigma^2}\right\}$) et les multi-quadratiques (calculées $\sqrt{\frac{r^2}{\sigma^2}+1}$). Le méta-modèle déterminé par le *Kriging* est la somme entre une fonction représentant la tendance globale des points dans la BD et un processus stochastique. Cette méthode relève plus des statistiques et permet d'avoir autour du méta-modèle un certain intervalle de confiance. La dernière méthode permet de classifier la base de données en deux ensembles, par exemple faisable ou non faisable, grâce à des hyperplans. En pratique, les données ne sont pas séparables linéairement, ce qui complexifie cette approche.

Dans notre mémoire, nous utiliserons l'échantillonnage *LCVT*. De plus, nous utiliserons une version améliorée du *RBF*, *tRBF* (pour *tuned RBF*), pour créer le méta-modèle au sein de la boucle externe. Celui-ci teste simplement toutes des combinaisons de paramètres pour *RBF* et choisit la meilleure à chaque itération. Ainsi, les méta-modèles sont générés sans que l'utilisateur spécifie le type de fonction à base radiale et les valeurs des hyper-paramètres qu'il faut utiliser. A chaque itération un unique point est rajouté dans la base de données contenant les points qui permettent de créer le méta-modèle (du *SBO*). Au commencement, la base de données est remplie avec $n_D + 1$

points évalués selon la véritable fonction à minimiser où n_D est le nombre de dimension du problème (donc, le nombre de paramètres). Le nombre d'itérations est fixé à $100-(n_D+1)$, afin d'avoir évalué 100 points au total par la véritable fonction.

3.2 AG utilisé

Comme détaillé dans la Section 1.4, il existe différentes configurations de cet algorithme. Cependant, dans notre mémoire, nous avons utilisé une certaine configuration de l'algorithme génétique imposée par des études réalisées avant ce mémoire par les concepteurs de Minamo afin d'obtenir un algorithme robuste et efficace.

En comparaison aux termes utilisés dans la Section 1.4, la valeur de fitness qui fait partie de l'AG est à minimiser, et non à maximiser. Celle-ci est également appelée *Global Objectif* (GO). Elle tient compte de la valeur de la fonction objectif et de la pénalité introduit par Deb [11] et reprise dans la Section 1.4. Notons que, de manière pratique, ceci implique une augmentation du GO pour tous les individus d'une population composée uniquement d'individus non-faisables, quand des individus faisables sont introduits.

En comparaison au schéma vu précédemment, la création de la nouvelle population est basique dans Minamo. En effet, un remplacement des parents par les enfants nouvellement créés, mutés et évalués est organisé, sans choisir les enfants et sans modifier la taille de la population. Ainsi, nous avons dû créer cette boîte dans le code. Cependant, une légère sélection est organisée quand même. En effet, un élitisme est réalisé lors de la génération des enfants puisque les deux meilleurs individus sont directement ajoutés dans la génération suivante. Ceci permet de décroître le meilleur GO durant l'exécution et de ne pas l'augmenter.

Concernant le croisement, son taux est fixé à 100% dans l'AG. Ainsi, tous les individus peuvent participer au croisement. Le croisement est réalisé par le principe du tournoi de Deb, expliqué précédemment. Le croisement est réalisé avec remise entre deux parents. Concernant la mutation, son taux de mutation n'est que de 1%. L'opérateur utilisé est la mutation de Gauss. Le nombre d'itérations est fixé à 100.

Pour finir, le paramètre qui nous importe le plus est la taille de la population. Une bonne pratique dans Minamo pour les problèmes avec peu de dimensions est de fixer le nombre d'individus à $100 \times n_D$ dans l'AG, où n_D est le nombre de dimension. Ainsi, un problème en $10D$ démarre avec 1000 individus et, avant ce travail, la taille demeurait constante.

3.3 Progrès désirés

Comme la taille de la population est constante et conséquente au sein de l'algorithme génétique, l'exécution est plus lente, surtout lorsque l'AG est utilisé seul pour résoudre la minimisation sur la véritable fonction. Ainsi, nous désirons améliorer ce paramètre avec différentes méthodes que nous avons introduites dans notre état de l'art. Nous espérons ainsi faire moins d'évaluations de fonctions au cours de l'exécution sans détériorer la recherche du minimum. Le progrès principal désiré est donc de réduire le temps d'exécution, en ayant une valeur de fitness semblable à Minamo avant notre intervention. Celui-ci est souhaité en particulier pour l'AG sans la boucle *SBO*. Le second progrès visé est d'améliorer cette recherche. Plus particulièrement lors d'une minimisation assistée par méta-modèles. C'est-à-dire que nous souhaitons obtenir de meilleurs résultats : se rapprocher du véritable minimum, quand il est unique.

3.4 Explication des six cas tests utilisés

Pour toutes les méthodes que nous avons implémentées (*Saw-Tooth*, *GAVaPS* et *FiScIS-EA*), nous les avons exécutées sur six cas tests communs afin de pouvoir tirer des conclusions entre elles et de sélectionner les meilleurs paramètres pour chacune d'elles. Ces six cas tests sont des fonctions de benchmark de la littérature. Nous avons : *Rosenbrock* en 2D et 10D, *Rastrigin 10D*, *G10*, *G7* et *G2* en 10D. Les trois premières sont des problèmes sans contraintes et les trois dernières sont des problèmes avec contraintes. Rappelons à présent leur description.

Rosenbrock

Pour *Rosenbrock*, son domaine est $x_i \in [-2, 2]\ \forall i = 1, \ldots, n_D$ où n_D est le nombre de dimension. Celui-ci est 2 pour *Rosenbrock 2D*, ou 10 pour ce problème en 10D. Son calcul est réalisé comme suit :

$$f(x) = \sum_{i=1}^{n_D-1} (1 - x_i)^2 + 100\ (x_{i+1} - x_i^2)^2.$$

Son minimum est à 0 quand $x^\star = (x_0, \ldots, x_{n_D}) = (1, \ldots, 1)$. De plus, *Rosenbrock 2D* est un problème uni-modale tandis que *Rosenbrock 10D* est un problème multi-modal [42].

Rastrigin

De même, *Rastrigin 10D* ($n_D = 10$) possède son minimum en 0. Cependant, elle possède énormément de minima locaux. Ainsi, le véritable défi est de ne pas rester coincé dans un puits local. Les coordonnées qui minimisent le problème sont $x^\star = (x_0, \ldots, x_{n_D}) = (0, \ldots, 0)$. La fonction est calculée comme suit :

$$f(x) = 10\ n_D + \sum_{i=1}^{n_D} x_i^2 - 10\ \cos(2\pi x_i)).$$

De plus, son domaine est $x_i^\star \in [-5.12, 5.12]\ \forall i = 1, \ldots, n_D$.

G7

Ensuite, le problème suivant est *G7*. Celui-ci possède dix variables et huit contraintes. Sa fonction à minimiser est la suivante :

$$\begin{aligned} f(x) \quad = \quad & x_1^2 + x_2^2 + x_1 x_2 - 14x_1 - 16x_2 \\ & + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2 \\ & + 2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 \\ & + 2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45, \end{aligned}$$

où $x_i \in [-10, 10]\ \forall i = 1, \ldots, 10$. Et ses contraintes sont :

$$\begin{aligned} c_1(x) \quad &\equiv \quad -105 + 4x_1 + 5x_2 - 3x_7 + 9x_8 \leq 0 \\ c_2(x) \quad &\equiv \quad 10x_1 - 8x_2 - 17x_7 + 2x_8 \leq 0 \\ c_3(x) \quad &\equiv \quad -8x_1 + 2x_2 + 5x_9 - 2x_{10} - 12 \leq 0 \\ c_4(x) \quad &\equiv \quad 3(x_1 - 2)^2 + 4(x_2 - 3)^2 + 2x_3^2 - 7x_4 - 120 \leq 0 \\ c_5(x) \quad &\equiv \quad 5x_1^2 + 8x_2 + (x_3 - 6)^2 - 2x_4 - 40 \leq 0 \\ c_6(x) \quad &\equiv \quad x_1^2 + 2(x_2 - 2)^2 - 2x_1 x_2 + 14x_5 - 6x_6 \leq 0 \\ c_7(x) \quad &\equiv \quad 0.5(x_1 - 8)^2 + 2(x_2 - 4)^2 + 3x_5^2 - x_6 - 30 \leq 0 \\ \text{et } c_8(x) \quad &\equiv \quad -3x_1 + 6x_2 + 12(x_9 - 8)^2 - 7x_{10} \leq 0. \end{aligned}$$

Pour finir, comme cette fonction possède beaucoup de minima, nous ne disposons que de la meilleure solution connue, qui est $x^\star = (2.17199, 2.36368, 8.77392, 5.09598, 0.99065, 1.43057, 1.32164, 9.82872, 8.28009, 8.37592)$ pour laquelle $f(x^\star)$ vaut 24.30620. Pour vérifier si les méthodes implémentées sont satisfaisantes, la valeur cible pour la fonction est $f(x^\star) = 25.0$.

G10

La fonction *G10*, qui est en 8D, est définie par :

$$f(x) = x_1 + x_2 + x_3,$$

où $x_1 \in [100, 10000]$, x_2 et $x_3 \in [1000, 10000]$. Cependant, pour complexifier la recherche, elle est sujette à six contraintes. Celles-ci sont :

$$\begin{aligned} c_1(x) &\equiv -1 + 0.0025(x_4 + x_6) \leq 0 \\ c_2(x) &\equiv -1 + 0.0025(x_5 + x_7 - x_4) \leq 0 \\ c_3(x) &\equiv -1 + 0.01(x_8 - x_5) \leq 0 \\ c_4(x) &\equiv -x_1 x_6 + 833.33252 x_4 + 100 x_1 - 83333.333 \leq 0 \\ c_5(x) &\equiv -x_2 x_7 + 1250 x_5 + x_2 x_4 - 1250 x_4 \leq 0 \\ \text{et } c_6(x) &\equiv -x_3 x_8 + 1250000 + x_3 x_5 - 2500 x_5 \leq 0 \end{aligned}$$

où $x_i \in [10, 1000]$ $\forall i = 4, \ldots, 8$. Notons que la meilleure solution connue est $x^\star = (579.30668, 1359.97067, 5109.97065, 182.01769, 295.60117, 217.98230, 286.41652, 395.60117)$ où $f(x^\star) = 7049.24802$. De plus, la valeur cible pour cette fonction est $f(x^\star) = 8000.0$.

G2 plog 10

Pour finir, le dernier problème à minimiser est *G2* en 10D. La définition de sa fonction est :

$$f(x) = -\left| \frac{\sum_{i=1}^{n_D} \cos^4(x_i) - 2 \prod_{i=1}^{n_D} \cos^2(x_i)}{\sqrt{\sum_{i=1}^{n_D} i x_i^2}} \right|,$$

où le domaine de définition est $x_i \in]0, 10]$, pour tout $i = 1, \ldots, n_D$, où $n_D = 10$. Il ne possède que deux contraintes. Celles-ci sont :

$$\begin{aligned} c_1(x) &\equiv \text{plog}\left(0.75 - \prod_{i=1}^{n_D} x_i\right) \leq 0 \\ \text{et } c_2(x) &\equiv \sum_{i=1}^{n_D} x_i - 7.5 n_D \leq 0 \end{aligned}$$

où la fonction *plog* est définie par Regis et Shoemaker [43] comme suit :

$$\text{plog}(\tilde{x}) = \begin{cases} \log(1 + \tilde{x}), & \text{si } \tilde{x} \geq 0 \\ -\log(1 - \tilde{x}), & \text{sinon,} \end{cases}$$

où $\tilde{x}$ est un réel quelconque et la fonction *log* est la fonction logarithmique en base 10. Cette fonction *plog* a comme particularité d'atténuer les fonctions extrêmement élevées ou négatives, d'être strictement croissante et d'être symétrique par rapport à l'origine. Notons que la meilleure solution connue pour ce problème d'optimisation est $f(x^\star) = -0.74063$. Cependant, la valeur cible est de -0.28.

Relations entre les cas tests et synthèse des paramètres

Notons que nous avons choisi différents types de fonctions. En effet, *Rosenbrock 2D* avec ses paramètres fixés est la fonction la plus simple que nous allons analyser, puisqu'elle ne possède qu'un seul minimum (uni-modale), qu'elle est quadratique et qu'elle n'est soumise à aucune contrainte.

Cependant, *Rosenbrock 10D* est une fonction multi-modale [42]. De plus, ces deux cas tests sont des fonctions non-convexes. Comme fonction sans contrainte, nous avons également *Rastrigin 10D* qui est une fonction objectif non linéaire et possédant énormément de minima locaux. Elle est d'ailleurs l'exemple typique pour les cas tests multi-modaux. Elle est ainsi non convexe. Ensuite, tous les problèmes avec contraintes sont des cas tests multi-modaux et non-convexes. Toutefois, *G7* est quadratique, *G10* est linéaire dans sa définition mais quadratique dans ses contraintes et *G2 plog 10* est non-linéaire. Toutes ces informations sont synthétisées dans le Tableau 3.1.

Problème	n_D	Convexe (oui/non)	uni/multi-modal	(non-)linéaire ou quadratique
Rosenbrock	2	non	uni-modale	quadratique
Rosenbrock	10	non	multi-modal	quadratique
Rastrigin	10	non	multi-modal	non-linéaire
G7	10	non	multi-modal	quadratique
G10	8	non	multi-modal	linéaire
G2 plog 10	10	non	multi-modal	non-linéaire

TABLE 3.1 – Ce tableau synthétise les types de cas tests utilisés, s'ils sont linéaires ou non, s'ils sont multi-modaux ou non, etc.

Pour finir, le Tableau 3.2 synthétise les valeurs fixées pour chaque paramètre selon le cas test. Ces paramètres sont la taille minimale autorisée (μ_{min}), la taille maximale (μ_{max}), la taille moyenne ($\overline{\mu}$), l'écart entre la taille moyenne et la taille maximale (D), le nombre de générations de l'AG (t_{max}), le nombre d'itérations en *SBO* ($\overline{t_{max}}$) et la taille de la BD en *SBO* ($\overline{n}$).

Problème	n_D	μ_{min}	μ_{max}	$\overline{\mu}$	D	t_{max}	$\overline{t_{max}}$	$\overline{n}$
Rosenbrock	2	10	200	105	95	100	97	3
Rosenbrock	10	50	1000	525	475	100	89	11
Rastrigin	10	50	1000	525	475	100	89	11
G7	10	50	1000	525	475	100	89	11
G10	8	40	800	420	380	100	91	9
G2 plog 10	10	50	1000	525	475	100	89	11

TABLE 3.2 – Ce tableau synthétise les valeurs fixées pour chaque paramètre selon le cas test.

Notons que tous ces problèmes sont à minimiser. Ainsi, aucune transformation ne doit être réalisée et le principe de Deb, utilisé dans Minamo, permettra d'intégrer les contraintes non-respectées dans le calcul du *GO*.

3.5 Explication des graphiques d'analyse

Au cours de notre mémoire, nous comparerons les méthodes que nous avons implémentées et modifiées dans Minamo, avec la version originale de Minamo. Pour ce faire, nous avons exécuté chacune des méthodes sur cent populations initiales différentes et en utilisant cent graines différentes pour les nombres aléatoires. Pour synthétiser les résultats de cent exécutions différentes et pour comparer les méthodes à Minamo, nous utiliserons certains graphiques que nous explicitons ci-dessous :

— des boîtes à moustache pour comparer la distribution de la solution finale obtenue pour chacune des méthodes ;

— des boîtes à moustache pour déterminer la distribution du temps d'exécution (en seconde) que prend chacune des méthodes implémentées ;
— des boîtes à moustache pour connaître le nombre d'évaluations de fonctions utiles selon chacune des méthodes (uniquement pour l'AG pur) ;
— des courbes de convergence moyenne pour chacune des méthodes ;
— des profils de données et de performance (pour l'AG intégré dans une boucle *SBO*).

boîtes à moustache

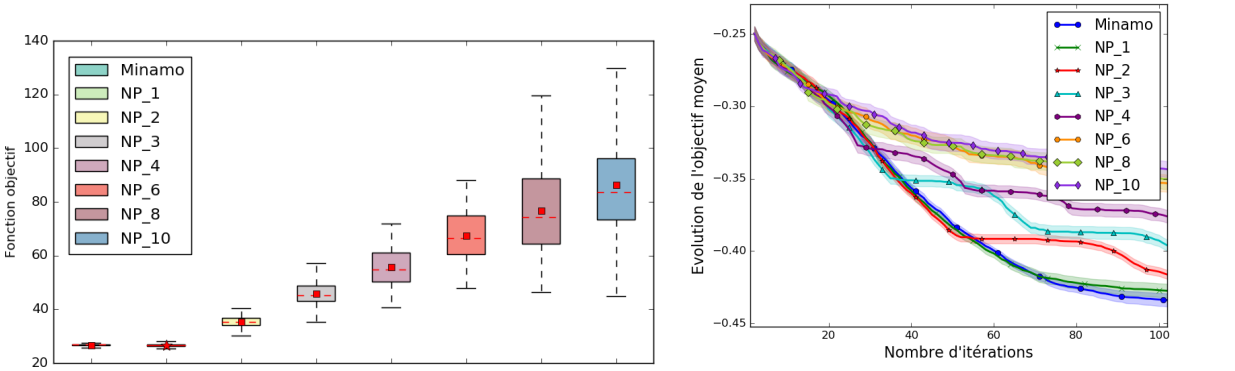
Chaque problème d'optimisation sont résolu par différentes méthodes, à partir de cent populations aléatoires différentes et cent graines différentes pour les nombres aléatoires. De plus, pour chacune d'entre elles, nous connaissons : leur temps d'exécution (en seconde), leur solution finale obtenue (en *log 10* pour *Rosenbrock 10D*) et le nombre total d'individus créés durant tout le processus et évalués par la véritable fonction objectif. Notons que cette dernière information ne concernera donc que les exécutions de l'AG pur (avec nos méthodes intégrées). Ces exécutions, au nombre de cent, ont des résultats assez représentatifs d'une exécution standard. Nous avons donc décidé de synthétiser ces résultats dans des boîtes à moustache afin de connaître leur distribution par rapport à Minamo.

Dans la Figure 3.3(a), nous montrons à titre d'exemple la distribution de la solution finale pour G7, selon différentes méthodes que nous avons implémentées, pour comparer avec Minamo. La médiane des résultats d'une boîte à moustache est représentée par un trait en pointillé dans la zone colorée. Par exemple, dans la boite de NP_10, la valeur médiane de la fonction objectif est proche de 82. La valeur moyenne de chaque cas test est représenté par un carré, et pour la meilleure moyenne celle-ci est représentée par une étoile. Ainsi, dans cette figure, la moyenne de NP_10 est plus élevée que la médiane et vaut aux alentours de 85, tandis que la meilleure moyenne est pour la boîte à moustache de NP_1. L'encadrement de la boite représente le premier et troisième quartile. Par contre les extrémités, les extrémités des moustaches, représentent les quantiles 1.5 et 98.5, sans afficher les valeurs qui sont considérées comme extrêmes, qui sont alors celles en dessous du quantile 1.5 ou au dessus du quantile 98.5.

La lecture de ces boîtes à moustache est faite de différentes manières selon la nature des données qui y sont représentées. Par exemple, pour les données sur la convergence finale (la valeur de l'objectif final), une version A de l'AG est dite meilleure qu'une version B si la boîte à moustache associée à A est plus basse que celle de la version B, car les fonctions objectifs doivent être minimisées. Il en est de même pour les boîtes à moustache qui recensent le temps d'exécution ou le nombre d'évaluations de fonctions. En effet, le temps d'exécution doit être réduit et le nombre d'évaluations de fonctions aussi.

Notons que les boîtes à moustache pour le nombre d'évaluations de fonctions ne sont utiles que pour l'AG pur. En effet, l'AG intégré au sein de la boucle *SBO* n'évalue pas la véritable fonction (coûteuse) puisqu'il minimise le méta-modèle de manière systématique.

Pour finir, les boîtes à moustache ne permettent pas de distinguer avec certitude, des fois, si une méthode est meilleure qu'une autre ou que Minamo. Dans un tel cas, il aurait fallut réaliser des tests statistiques afin de vérifier que la moyenne (ou la médiane) des valeurs des fonctions objectifs (par exemple) pour une version était bien différente que la moyenne des autres versions. Faute de temps, nous réaliserons de tels tests pour la défense orale de ce mémoire.



(a) boîtes à moustache pour la solution finale pour *G7*

(b) Courbes de convergence moyenne pour *G2 plog 10*

FIGURE 3.3 – Exemple de boîtes à moustache et de courbes de convergence moyenne générées.

Courbes de convergence moyenne

Pour afficher la convergence au cours des itérations, nous avons opté pour un graphique qui utilise la moyenne des cent itérations et affiche un intervalle de confiance à 95% autour de la courbe de convergence de chaque cas test. Cet intervalle de confiance s'exprime comme suit :

$$I_C = \frac{\delta}{\sqrt{n_{stat}}},$$

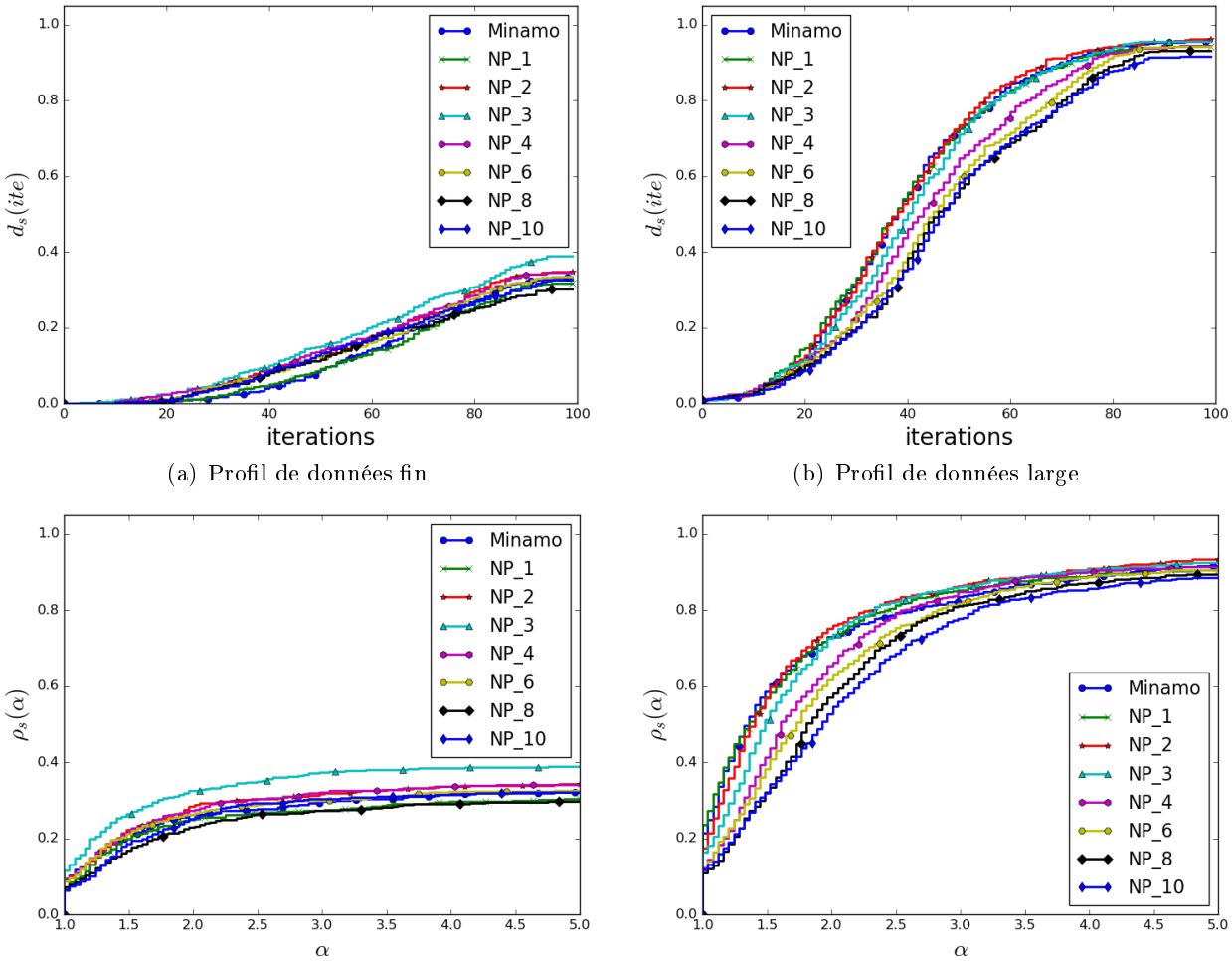
où n_{stat} représente le nombre d'exécutions de l'algorithme avec n_{stat} graines aléatoires différentes ($n_{stat} = 100$) et δ est l'écart-type des résultats. Dans la Figure 3.3(b), nous avons la convergence vers les solutions obtenues pour chaque version avec l'intervalle de confiance à 95%. Il arrive que l'intervalle de confiance soit peu étendu, comme ici, ce qui signifie que, pour une méthode donnée, celle-ci a le même comportement au cours des itérations pour chacune des exécutions. Par contre, il se peut que deux courbes s'entremêlent avec leur intervalle de confiance, comme la méthode NP_6 et NP_8 ici. Ceci signifie donc que les deux versions ont un comportement semblable durant les itérations. Notons que l'ordonnée de ces graphiques représentent bien souvent la valeur des fonctions objectifs où nous avons appliqué un log en base 10 (sauf pour *G2 plog 10*), pour mieux faire ressortir les différences entre les méthodes.

Profils de données et de performance

Pour finir, les profils de données et de performance sont deux types de graphiques introduits par Dolan et More [44]. Un exemple de ces graphiques a été généré dans la Figure 3.4. Il s'agit de synthétiser les résultats de plusieurs cas tests en un seul graphique, résolus avec différentes méthodes d'optimisation. Le but est d'avoir une vue d'ensemble sur l'état de la résolution des différents cas tests. Ils partent de l'idée qu'un problème est résolu s'il atteint une certaine valeur, les valeurs cibles décrites dans la section précédente pour chaque cas test. Cependant, la notion de faisabilité est absente dans cette approche, alors nous partons du principe qu'une exécution a résolu le problème

1. si la solution heuristique renvoyée par l'algorithme est inférieure à la valeur cible et
2. si le point associé à cette solution est faisable (respecte toutes les contraintes).

Le seuil va être fixé de deux manières différentes : un seuil grossier (large) et un seuil pointilleux (fin). Le premier permet de savoir si globalement les solutions trouvées sont admissibles, tandis que le second appuie sur le fait que les solutions calculées ont considérablement bien convergé vers la



(a) Profil de données fin

(b) Profil de données large

(c) Profil de performance fin

(d) Profil de performance large

FIGURE 3.4 – Exemple de graphiques de profil de données et de performance générés à partir des six cas tests présentés dans la section précédente, pour différentes méthodes.

meilleure solution connue.

Le premier graphique est le profil de données (Figures 3.4(a) et 3.4(b)). Celui-ci indique le pourcentage de cas tests qui ont été résolus selon le seuil pour chaque itération. Par exemple, dans la Figure 3.4(a), 40% des cas tests ont été résolus de manière fine par la stratégie NP_3 après cent itérations.

Le second graphique est le profil de performance (Figures 3.4(c) et 3.4(d)). Celui-ci est souvent utilisé dans la littérature pour comparer des méthodes entre elles. Le but d'un tel graphique est de connaître le meilleur algorithme au début de la recherche de minima et à la fin, selon l'utilisation d'un budget autorisé d'évaluations de fonctions. L'ordonnée d'un tel graphique représente le pourcentage des cas tests résolus selon un coefficient multiplicateur de ce budget (abscisse), noté α. Ainsi, quand $\alpha = 1$ dans nos graphiques, peu d'évaluations de fonctions ont déjà été réalisées. A l'opposé, quand $\alpha = 5$, toutes les évaluations de fonctions ont été épuisées (comme le budget est fixé en *SBO* à 100). Deux lectures peuvent alors être faites sur un tel graphique. Une courbe située au dessus d'une autre en un α fixé signifie qu'elle est plus performante (selon ce α). Ainsi, quand $\alpha = 1$, la courbe la plus performante est synonyme d'efficacité. En effet, cela signifie que l'algorithme associé à cette courbe résout rapidement les cas tests. Puis, quand $\alpha = 5$, la méthode associée à une courbe située au dessus d'une autre est dite plus robuste, car elle permet de résoudre un plus grand nombre de

problèmes pour un nombre d'évaluations de fonctions donné. Par exemple, la Figure 3.4(c) montre que NP_3 est la version la plus efficace et la plus robuste pour un seuil fin. La Figure 3.4(d) montre que Minamo est plus efficace que NP_4, tandis que NP_4 est légèrement plus robuste que Minamo.

Notons que nous n'avons pas pu générer de graphes similaires pour les problèmes résolus avec l'AG pur, puisque cette approche compare des résultats entre des algorithmes qui évoluent selon le même nombre d'évaluations de fonctions. Ainsi, ces graphes sont générés uniquement pour les cas tests résolus avec l'AG intégré dans une boucle *SBO*.

Maintenant que nous avons présenté Minamo, les différents cas tests et les différents graphes d'analyse, nous pouvons décrire les méthodes que nous avons implémentées dans Minamo. Nous commencerons dans le chapitre suivant par la méthode *Saw-Tooth*.

Chapitre 4

Saw-Tooth - explication, implémentation et résultats

Dans ce chapitre, nous décrirons en détail l'algorithme génétique *Saw-Tooth* proposé par Koumousis et Katsaras [1]. Ce réglage appartient à la famille des contrôles déterminés, car il ne tient pas compte de l'avancement de la recherche de la solution. Bien que les formules peuvent sembler plus complexes que la philosophie associée, cette méthode est la plus simple à intégrer dans l'implémentation existante. En effet, elle a nécessité peu de maintenance et nous n'avons pas été obligés de mettre en place des astuces pour garantir son bon déroulement, en comparaison des autres méthodes, comme nous le verrons par la suite.

Nous réaliserons une première section où sera développée la méthode originale, avec ses formules, ses paramètres déterminés dans l'article et sa philosophie. Ensuite, la seconde section concernera son implémentation dans Minamo. Nous y présenterons la manière dont nous avons implémenté *Saw-Tooth* dans Minamo ainsi que de légères adaptations de celui-ci. Après cela, que ce soit avec l'AG pur ou avec la boucle *SBO*, nous réaliserons une étude des paramètres ajoutés par cette méthode afin de déterminer quelles sont leurs valeurs adéquates. Cela nous fournira une comparaison entre Minamo de base et *Saw-Tooth* avec les meilleurs paramètres sur des cas tests exécutés sur l'AG pur puis avec l'AG intégré dans une boucle *SBO*.

4.1 Méthode originale

Koumousis et Katsaras [1] introduisent leur méthode en soulignant l'importance d'avoir une stratégie qui décroît le nombre d'individus présents dans la population, tout en l'augmentant lorsque la taille minimale de la population est atteinte.

Pour rajouter des individus dans une population générale, Koumousis et Katsaras [1] proposent d'en créer des nouveaux aléatoirement dans l'espace. Ceci permet d'introduire des nouveaux gènes dans la population. De plus, pour des problèmes multi-modaux, c'est-à-dire avec beaucoup de minima locaux, cette stratégie permet de sortir d'un puits en redonnant de la variabilité, comme l'opérateur de mutation le fait. En effet, la mutation est un processus plus lent qui touche certains gênes de la population, tandis que la réinitialisation de la population est plus drastique et concerne un sous-ensemble de la population. En réalité, Minamo possède un faible taux de mutation. Des études internes sont à réaliser pour modifier cette valeur, en l'augmentant et la fixant ou en la modifiant aussi au cours des itérations. En revanche, le fait de faire varier la population possède l'avantage de débuter avec une population contenant beaucoup d'individus. Ceci apporte ainsi beaucoup de variabilité, c'est-à-dire d'exploration de l'espace d'entrée, et est donc bénéfique. Puis, en diminuant la taille de la population, l'algorithme se concentre sur de bons individus, essayant ainsi de converger dans un minimum.

Pourtant, cette stratégie ne nécessiterait pas plus d'évaluations de fonctions pour mieux fonctionner. En effet, dans leur article [1], ils ont réalisé une courte étude pour opposer trois tests qui évoluent sur 200 générations. Pour le premier test, ils ont fixé la taille de la population à 100. Pour le second test, la taille décroît linéairement au cours des itérations. La taille initiale vaut 150 tandis que la taille finale vaut 50. Pour le troisième test, ils décroissent la taille de la population linéairement en débutant à une taille initiale valant 195 et finissant à 5. Ainsi, ces trois stratégies auront toutes évaluées 20′000 fois la fonction. Leur constat affirme leur intuition : le troisième test est le meilleur. En effet, commencer avec plus d'individus et finir avec moins d'individus est bénéfique.

Ainsi, pour réaliser un algorithme qui combine ces deux méthodologies (décroissance et introduction de nouveaux individus), Koumousis et Katsaras [1] décident de combiner une décroissance linéaire durant un certain nombre d'itérations T et une augmentation ponctuelle de la population en injectant des nouveaux individus. Pour ce faire, à chaque itération t, ils calculent le nombre d'individus $\mu(t)$ qui doivent composer la population pour cette génération. Ce nombre est calculé par la formule suivante :

$$\mu(t) = \left\lfloor \overline{\mu} + D - \frac{2D}{T-1}\left(t - T\ \left\lfloor \frac{t-1}{T} \right\rfloor - 1\right)\right\rfloor, \tag{4.1}$$

où $\overline{\mu}$ représente le nombre d'individus moyen au cours d'une période de T itérations, D est l'amplitude de la taille de la population et $\lfloor . \rfloor$ est l'opérateur prenant la partie entière d'un réel. En fixant $\overline{\mu}$ à 30, D à 10 et T à 10, comme dans la Figure 4.1, nous obtenons quand $t = 1$ une taille souhaitée de $\overline{\mu} + D$, soit 40 individus. Ensuite, quand $t = T$, la taille calculée est de 20 car $\mu(T) = \overline{\mu} - D$. Pour finir, à l'itération suivante, la taille est remise à l'origine, soit à 40 individus.

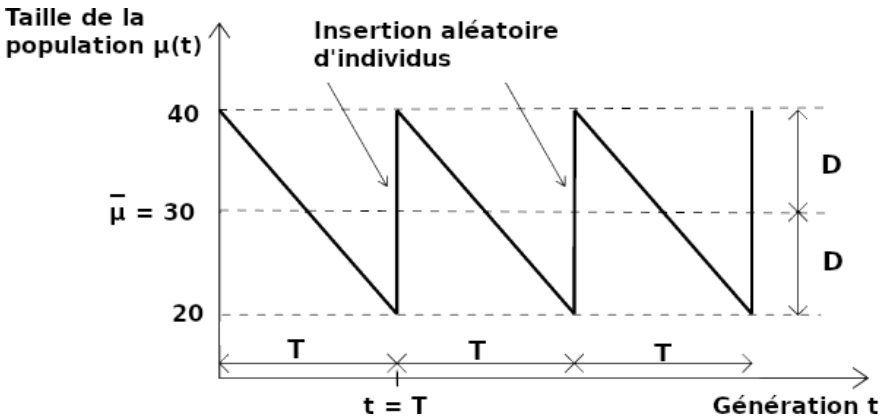


FIGURE 4.1 – Ce schéma représente l'évolution de la taille de la population au cours des générations selon la formule de *Saw-Tooth* où $\overline{\mu}$ est fixé à 30 et D est égal à 10. Il est inspiré de l'article de Koumousis et Katsaras [1].

La philosophie dans leur article est de fixer la taille moyenne, $\overline{\mu}$, et l'écart entre $\overline{\mu}$ et la taille maximale. A l'opposé, nous fixons la taille maximale et minimale voulue, puis nous calculons la taille moyenne et l'écart. En effet, nous désirons que, pour la plupart de nos méthodes, la taille de population initiale soit égale à cent fois le nombre de paramètres. De plus, la taille de la population ne pourra jamais descendre en dessous de cinq fois le nombre de paramètres afin de réaliser une bonne décroissance, comme dans l'article cela semblait être intéressant. Ainsi, nous avons :

$$\begin{cases} \mu_{min} &= 5 \times n_D \\ \mu_{max} &= 100 \times n_D \end{cases}$$

où n_D est le nombre de dimension, μ_{min} est la taille minimale autorisée et μ_{max} est la taille maximale autorisée. Par ces relations, nous pouvons calculer $\overline{\mu}$ et D de la manière suivante :

$$\begin{cases} D &= \dfrac{\mu_{max} - \mu_{min}}{2} \\ \overline{\mu} &= \mu_{max} - D. \end{cases}$$

Pour finir, le calcul de T est réalisé en fixant le nombre de périodes ($\overline{T}$) souhaité durant l'exécution des cent itérations. Ainsi,

$$T = \left\lfloor \frac{100}{\overline{T}} \right\rfloor,$$

où $\lfloor . \rfloor$ est la partie entière. Par ces relations, nous constatons que nous devrons uniquement déterminer le nombre de périodes $\overline{T}$ le plus adapté selon le type de problèmes que nous allons résoudre.

Dans l'article [1], Koumousis et Katsaras ont réalisé une analyse de paramètres sur des fonctions objectifs uni-modales et des fonctions multi-modales. A chaque fois, ils ont lancé trente exécutions (*runs*) de l'algorithme avec des graines de nombre aléatoires (*seeds*) différentes pour chaque combinaison de paramètres. Comme conclusion pour les problèmes uni-modaux, ils ont remarqué que :

1. la taille de la population moyenne $\overline{\mu}$ intéressante est de 40 individus ;
2. $\frac{T}{\overline{\mu}} = 0.40$ (ainsi T doit valoir 16 itérations) ;
3. $\frac{D}{\overline{\mu}} = 0.85$ (ainsi D doit valoir 34 individus).

Pour les problèmes multi-modaux, ils concluent que :

1. $\overline{\mu}$ doit valoir 60 individus ;
2. $\frac{T}{\overline{\mu}} = 0.70$ (ainsi T doit valoir 42 itérations) ;
3. $\frac{D}{\overline{\mu}} = 0.99$ (ainsi D doit valoir 59 individus).

Ainsi, nous constatons que les problèmes uni-modaux ont besoin d'un plus grand nombre de dents de scie que les problèmes multi-modaux. Cela peut paraître étrange, car normalement les problèmes uni-modaux ne possèdent qu'un minimum, donc il ne serait pas à priori nécessaire de faire varier la taille de la population. Pourtant, nous aurons un résultat similaire dans les prochaines sections. Ceci peut probablement s'expliquer sur le fait que, pour les problèmes avec plusieurs minima locaux, il est difficile de descendre profondément dans un minimum local si nous ne réalisons pas assez d'itérations. Notons que les valeurs présentées précédemment ont été obtenues en fixant le taux de croisement à 0.90 alors que, dans Minamo, il est fixé à 1. Ainsi, il est primordial de tester les valeurs selon les relations que nous avons déterminé précédemment. Mais avant de les déterminer, décrivons l'implémentation de cette méthode dans la section suivante.

4.2 Implémentation

D'un point de vue du l'implémentation dans Minamo, nous avons décidé d'introduire la gestion de la population avant le croisement et la mutation, dans la fonction qui contient la boucle d'itération. Ce choix nous paraît le plus judicieux pour plusieurs raisons. Premièrement, cette stratégie pourra directement réduire ou augmenter la population au bon moment. En effet, cela détermine le nombre d'enfants à créer et permet de garder les bons parents avant de générer les enfants. Deuxièmement pour réaliser cette sélection, il est nécessaire d'avoir les valeurs de fitness pour tous les individus de la population. Ceci ne peut être fait si nous réduisons les individus après la génération des enfants, car l'évaluation des individus se fait en dehors de cette fonction. Pour finir, l'implémentation des autres stratégies que nous allons étudier pourront être ajoutées au même emplacement.

Pour diminuer le nombre d'individus dans la population, nous réalisons une sélection par tournoi inverse. C'est-à-dire que nous réalisons un tournoi qui renvoie le pire individu lorsque deux individus sont comparés entre eux. Ainsi de moins bons individus peuvent être gardés dans la population. Ceci garantit une certaine variabilité dans les individus, qui peut être bénéfique pour les itérations suivantes, car cela peut permettre de sortir de minima locaux.

Pour ajouter aléatoirement des individus dans la population (afin de la combler), nous utilisons l'échantillonnage par hypercubes latins (*Latin Hypercube Sampling*) que nous avons décrit dans la Section 3.1. En plus d'introduire des individus aléatoirement dans l'espace, cette stratégie permet de bien les distribuer, afin d'avoir une bonne variabilité dans les gênes. Notons qu'après avoir introduit de nouveaux individus, nous ne pouvons pas réaliser le restant de la boucle de l'AG puisque ces individus ne sont pas évalués et qu'ils ne pourront jamais être sélectionnés lors du tournoi propre au croisement. Malheureusement, une solution simple aurait été de demander leur évaluation une fois créés. Cependant, cela n'était pas possible, car l'évaluation ne pouvait être fait qu'en dehors de la partie modifiée de l'implémentation et sinon il aurait fallut modifier l'architecture de Minamo.

Ainsi, l'Algorithme 1 synthétise la méthodologie utilisée et l'AG obtenu. Dans ce pseudo-code, nous avons traduit la nécessité de réévaluer la population après l'avoir complétée avec la stratégie LHS. Notons que les lignes 3. à 5. ne doivent plus être parcourues.

Algorithme 1 : *Saw-Tooth*

```
1  début
2      t = 1 ;
3      initialiser P(t) avec une taille μ_max ;
4      évaluer P(t) ;
5      tant que t ≤ 100 faire
6          calculer le nombre d'individus μ(t) à obtenir selon L'Équation 4.1 ;
7          si t > 1 et μ(t) < μ(t − 1) alors
8              enlever μ(t − 1) − μ(t) individus au sein de P(t) avec des tournois inverses entre
                deux individus ;
9          sinon si t > 1 et μ(t) > μ(t − 1) alors
10             ajouter μ(t) − μ(t − 1) individus au sein de P(t) avec la stratégie LHS ;
11             recommencer en 6 ;
12         croiser et générer un certain nombre d'enfants E(t) ;
13         muter ces enfants E(t) ;
14         P(t + 1) ← E(t) ;
15         t ← t + 1 ;
```

Maintenant que nous connaissons l'algorithme utilisé, nous pouvons déterminer les valeurs de l'unique paramètre retenu, qui est le nombre de périodes souhaitées, pour l'algorithme génétique pur et pour l'algorithme génétique intégré dans la boucle *SBO*. Nous allons ainsi exécuter cette méthode avec différentes valeurs pour $\overline{T}$ sur différents cas tests. Ceux-ci ont été décrits dans la Section 3.4.

4.3 Résultats pour l'AG

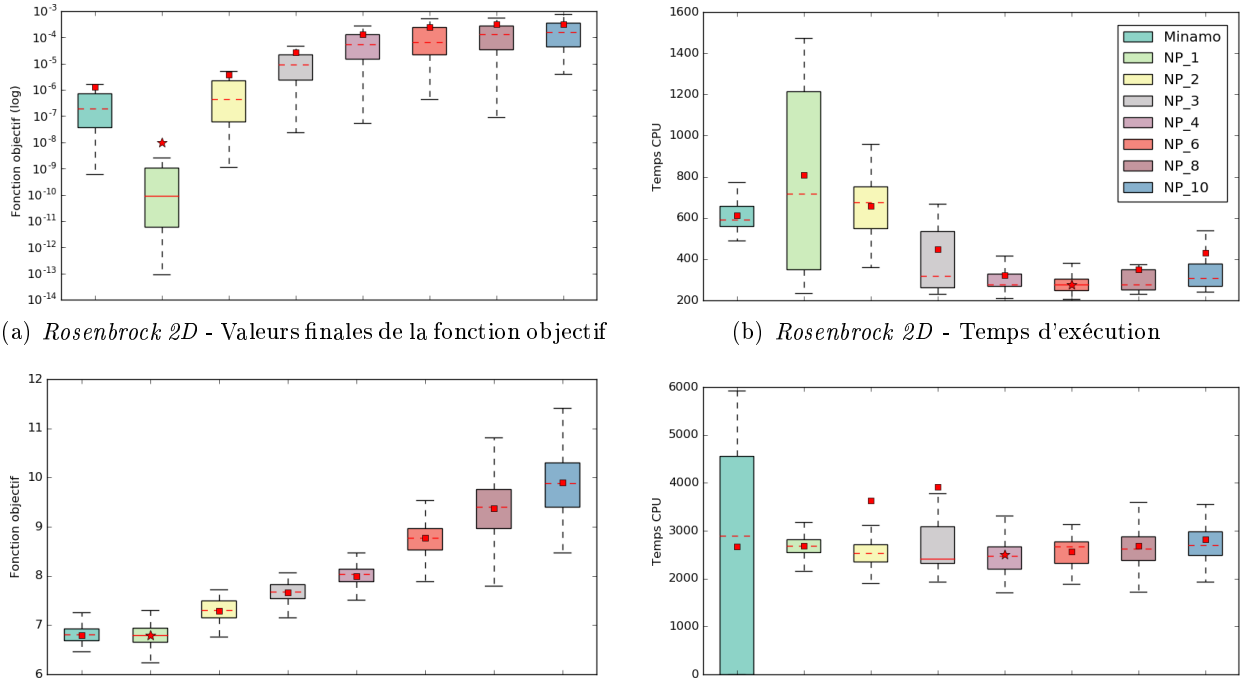
Pour réaliser cette section, nous avons résolu les six cas tests sur neuf versions. De plus, chaque version a été exécutée cent fois. Comme version, nous avons la version de Minamo avec la taille fixée pour toutes les itérations à μ_{max}. Mathématiquement, pour tout t entre 1 et 100, nous avons que $\mu(t) = \mu_{max}$ qui est calculé comme $100 \times n_D$. Avec ceci, sept versions de *Saw-Tooth* avec différentes

valeurs pour le nombre de périodes ($\overline{T}$) ont également été comparées. Ces valeurs ont été choisies parmi les entiers de 1 à 10, à savoir 1, 2, 3, 4, 6, 8 et 10. Dans cette section, nous utiliserons les labels suivants pour chacune des méthodes :

— Minamo : version de Minamo avec la taille fixée pour toutes les itérations à μ_{max} ;
— NP_1 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 1$;
— NP_2 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 2$;
— NP_3 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 3$;
— NP_4 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 4$;
— NP_6 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 6$;
— NP_8 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 8$;
— NP_10 : version de *Saw-Tooth* intégrée dans Minamo avec $\overline{T} = 10$.

Pour le problème classique uni-modal, *Rosenbrock 2D*, la Figure 4.2 représente ses résultats. D'abord, la version de *Saw-Tooth* la plus efficace est celle qui ne contient qu'une dent, c'est-à-dire une unique période. En effet, pour cette version, 50% de ses exécutions fournissent une solution de l'ordre de 9.25e-11, tandis que, pour Minamo, 50% de ses solutions sont de l'ordre de 1.99e-07. Ceci démontre que, pour ce cas test, cette méthode avec $\overline{T}$ qui vaut 1 fournit une solution heuristique plus proche de la solution réelle que Minamo, comme la solution réelle est en zéro. Ensuite, nous constatons que, plus $\overline{T}$ croît, plus les solutions médianes trouvées augmentent (par exemple, la Figure 4.2(a)). Ceci nous fait penser que, pour des problèmes uni-modaux, une unique décroissance linéaire de la taille de la population est largement suffisante. Une dernière observation, pour ce cas test, est que Minamo avec une taille de la population fixée à μ_{max} durant toutes les itérations est moins efficace que la version de *Saw-Tooth* qui ne posséderait qu'une période. L'avantage de *Saw-Tooth* avec $\overline{T} = 1$ est que nous avons eu besoin de la moitié moins d'individus par rapport à Minamo. En effet, $\mu_{moy} \approx \frac{\mu_{max}}{2}$ et, pour toutes les versions de *Saw-Tooth*, nous avons que le nombre d'individus créés au total est calculé par $t_{max} \times \mu_{moy}$ tandis que, pour Minamo, ce nombre est déterminé par $t_{max} \times \mu_{max}$. Concernant le temps d'exécution de chacune de ces méthodes, nous observons sur la Figure 4.2(b) que les versions pour *Saw-Tooth* avec $\overline{T}$ qui vaut 1 ou 2 sont plus lentes que Minamo et les autres versions de *Saw-Tooth*, bien que le nombre d'individus soit réduit par rapport à Minamo. Il est probable que le traitement de cette méthode alourdit légèrement l'exécution et que le gain en utilisant moins d'individus devient négligeable. En effet, la fonction *Rosonbrock 2D* à évaluer n'est pas la plus lourde en temps de calcul. Pour les autres cas tests, nous obtiendrons de meilleurs résultats concernant le temps.

Des conclusions semblables peuvent être réalisées pour les cinq cas tests restants, en utilisant la Figure 4.2 et la Figure 4.3. Premièrement, les boîtes à moustache concernant les valeurs finales pour les fonctions objectifs montrent les mêmes résultats que pour *Rosenbrock 2D*. En effet, les résultats sont de moins en moins bons quand $\overline{T}$ augmente. Ceci est vérifié pour les problèmes avec contraintes ou sans contraintes, et pour des problèmes linéaires ou non-linéaires. Deuxièmement, la version de *Saw-Tooth* avec $\overline{T} = 1$ donne des résultats ayant des valeurs plus petites que ceux renvoyés par Minamo. C'est-à-dire que les résultats de la méthode nouvellement implémentée améliore la recherche d'optimum. Notons que pour le problème *G2 plog 10* (Figure 4.3(g)), la recherche n'est pas améliorée mais elle n'est pas détériorée non plus. Dernièrement, une réduction du temps d'exécution de l'AG est à souligner pour les problèmes qui demandent beaucoup de temps de calcul pour évaluer un point. En effet, ceux-ci voient leur temps d'exécution être divisé par deux (pour *G7* et *G2 plog 10*, dans la Figure 4.3(d) et Figure 4.3(h) respectivement), voire par trois (pour *Rastrigin 10D* dans la Figure 4.3(b)), comme le nombre d'individus diminue en moyenne de moitié. Nous imaginons bien que, pour des problèmes industriels, comme ceux que Cenaero traite, cette méthode peut être bénéfique pour la recherche de solutions et pour la rapidité. Par contre, pour le problème *G10* (Figure 4.3(f)), le temps d'exécution ne semble pas être amélioré. Cependant, il semblerait que quelques anomalies sur le temps d'exécution soient survenues lors de l'exécution de



(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif

(b) *Rosenbrock 2D* - Temps d'exécution

(c) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

(d) *Rosenbrock 10D* - Temps d'exécution

FIGURE 4.2 – Ces graphiques représentent les résultats synthétisés des cas tests *Rosenbrock* (en 2D et 10D) résolus avec la méthode *Saw-Tooth* et la version de Minamo avec l'AG pur. Les graphiques de gauche représentent la variabilité des valeurs finales trouvées. Ceux de droite représentent les statistiques sur le temps d'exécution en seconde de chacune des versions. La même légende est utilisée pour les quatre figures.

certains tests. Par exemple, pour la version compilé de *Saw-Tooth* avec $\overline{T}$ valant 3 qui a été exécuté sur le problème *G2 plog 10* (Figure 4.3(h)). Il en est de même pour la résolution de *Rosenbrock* en 10D résolue avec Minamo (Figure 4.2(d)). Il se peut entièrement qu'un évènement extérieur à nos exécutions ait ralenti certains tests. Mise à part cela, le temps d'exécution semble bien être réduit selon la plupart des problèmes d'optimisation.

En plus de regarder les résultats finaux des différentes versions sur les différents problèmes d'optimisation, nous devons également regarder l'évolution de la convergence des différentes versions selon les différents cas tests. Ceci est montré dans la Figure 4.4. En effet, il peut être dangereux de conclure prématurément que la version de *Saw-Tooth* avec $\overline{T} = 1$ possède une meilleur convergence que Minamo, en regardant simplement les résultats finaux : il faut regarder l'évolution de la recherche de solution et s'assurer que Minamo converge moins rapidement aussi. Heureusement, la version de *Saw-Tooth* avec $\overline{T} = 1$ possède bel-et-bien une meilleure convergence. En effet, pour *Rosenbrock* en 2D qui possède une solution globale en zéro, cette version de *Saw-Tooth* plonge vers du 10e-8 au fur et à mesure des itérations, tandis que Minamo reste à la traîne (Figure 4.4(a)). Concernant les autres problèmes d'optimisation Minamo est légèrement moins performant que cette version de *Saw-Tooth*, mais eux deux sont bien meilleurs que *SawTooth* avec $\overline{T} > 1$.

Remarquons avant de conclure que certains temps d'exécution semblent avoir été atteint par un événement externe à nos exécutions. Par exemple, le temps d'exécution de Minamo pour le problème *Rosenbrock 10D* (Figure 4.2(d)) est fort variant. Il en est de même pour la version où $\overline{T} = 1$ pour *Rosenbrock 2D* (Figure 4.2(b)) et pour *G10* (Figure 4.3(f)). Pour finir, la même observation peut

être faite pour *G2 plog 10* (Figure 4.3(h)) pour la version $\overline{T} = 3$. Pour argumenter notre intuition, nous rappelons que toutes ces versions évaluent les fonctions objectifs le même nombre de fois. La gestion de la taille de la population se fait par un simple calcul. Pour finir, s'il devait y avoir un effet visible dans le temps d'exécution, il évoluerait selon le nombre de périodes $\overline{T}$. Or ce n'est pas le cas.

En conclusion de cette section, nous avons codé une méthode *Saw-Tooth* avec $\overline{T} = 1$ qui est deux fois plus rapide que Minamo et qui possède d'aussi bons résultats que Minamo, voire également de meilleurs résultats. Cependant, les utilisateurs finaux du logiciel utilisent rarement l'algorithme génétique pur. En effet, ceux-ci l'exécutent au sein d'une boucle *SBO* pour éviter une lourdeur d'exécution causée à l'évaluation de fonctions, qui est coûteuse. Ainsi, dans la section suivante, nous allons vérifier que de telles améliorations peuvent être observées grâce à ce réglage de paramètre.

4.4 Résultats pour l'AG intégré dans la boucle *SBO*

Dans cette section, nous utilisons les mêmes versions compilées de *Saw-Tooth* et de Minamo que la section précédente. De plus, nous garderons également les mêmes labels. Nous avons donc exécuté ces versions sur l'optimisation assistée par modèles de substitution contenant l'AG. Les résultats synthétiques concernant les valeurs finales des fonctions objectifs des six cas tests sont repris dans la Figure 4.5 et les résultats synthétiques exposant les temps d'exécution sont repris dans la Figure 4.6.

D'abord, selon la Figure 4.5, nous observons que, pour trois cas tests sur six, plus la valeur de $\overline{T}$ augmente, moins bons sont les résultats pour les versions de *Saw-Tooth*. Ceci conforte la conclusion réalisée dans la section précédente. Ces problèmes d'optimisation sont *Rosenbrock 2D* (pour lesquels les *outliers* augmentent dans la Figure 4.5(a)), *G7* (Figure 4.5(d)) et *G10* (Figure 4.5(e)), qui sont des problèmes linéaires ou quadratiques. Par contre, une telle observation n'est pas faite pour les problèmes contenant du cosinus dans la définition (*Rastrigin 10D* pour la Figure 4.5(c) et *G2 plog 10* pour la Figure 4.7(f)). En effet, il ne semblerait pas avoir de lien entre la valeur de $\overline{T}$ et les résultats obtenus. De plus, pour ces problèmes non-linéaires, il semblerait même que la version de *Saw-Tooth* avec $\overline{T} = 3$ est la meilleure. Ainsi, la version de *Saw-Tooth* où $\overline{T} = 1$ semble être celle à garder pour les problèmes qui sont linéaires ou quadratiques, tandis que la version de *Saw-Tooth* où $\overline{T} = 3$ devrait être gardée pour les problèmes non-linéaires.

Ensuite, selon la Figure 4.6, nous pouvons également conclure que le temps d'exécution est considérablement réduit pour tous les cas tests en utilisant *Saw-Tooth*. Il est même divisé par deux pour la version avec une dent ($\overline{T} = 1$). Cependant, la durée d'exécution augmente plus il y a de dents. Ceci est probablement dû à la génération aléatoire de nouveaux individus dans l'espace, qui peut être longue. Nous retenons donc la version de *Saw-Tooth* avec $\overline{T} = 1$, qui divise le temps d'exécution par deux.

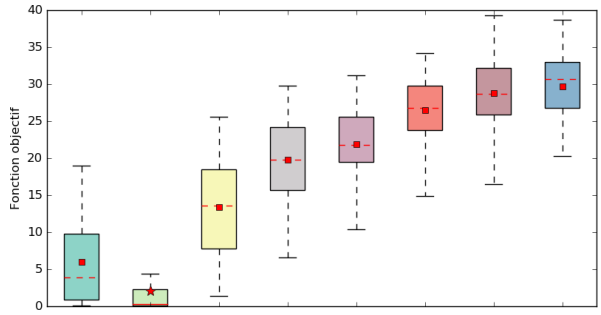
De plus, les graphiques de convergence repris dans la Figure 4.7 montrent que, pour Rosenbrock en 2D (Figure 4.7(a)), *Saw-Tooth* avec $\overline{T} = 1$ possède sans aucun doute une belle convergence par rapport à Minamo ; dans le sens qu'à chaque itération, cette nouvelle version possède en moyenne un meilleur individu que Minamo. Cependant, pour les problèmes non-linéaires qui sont *Rastrigin 10D* et *G2 plog 10*, *Saw-Tooth* avec $\overline{T} = 3$ converge mieux que *Saw-Tooth* avec $\overline{T} = 1$ et que Minamo.

La Figure 4.8 reprend les profils de données et de performance selon l'approche de Dolan et More [44]. Pour les profils de données, nous constatons que la version de Minamo est noyée parmi les versions implémentées. Cependant, *Saw-Tooth* avec $\overline{T} = 3$ résout le plus de problèmes, pour toutes les itérations (Figure 4.8(a)). De plus, d'un point de vue de la performance avec un seuil fin (Figure 4.8(c)), cette version est également bien meilleure que les autres variantes de *Saw-Tooth* et

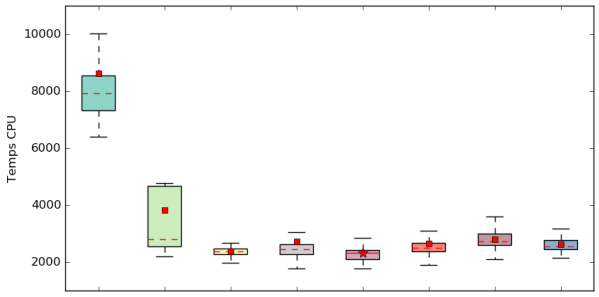
même de Minamo. Elle est ainsi efficace et robuste. Nous retiendrons cette version pour le chapitre comparatif de fin.

4.5 En résumé

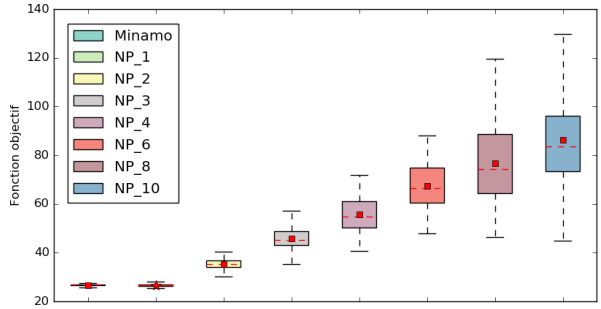
Après avoir décrit la méthode *Saw-Tooth* avec ses paramètres selon l'article de Koumousis et Katsaras [1], nous avons ramené le nombre de paramètres ajoutés à un. Celui-ci est le nombre de dents $\overline{T}$ à réaliser durant les cent itérations de l'AG. Pour rappel, une dent est une décroissance linaire de la taille de la population, en partant de μ_{max} pour arriver à μ_{min}, suivi d'une augmentation à la taille maximale. Dans notre cas μ_{max} valait la taille originale (soit $100 \times n_D$, où n_D est le nombre de dimension du problème considéré) et $\mu_{min} = 5 \times n_D$. Après avoir implémenté cette méthode dans Minamo avec sept valeurs différentes pour $\overline{T}$ (donnant sept versions), nous les avons exécutées sur six cas tests différents. D'abord, une analyse sur le comportement de ces versions au sein de l'algorithme génétique pur a été réalisée, suivi de celui-ci intégré dans une boucle *SBO*. De manière très concluante, les versions de *Saw-Tooth* avec $\overline{T} = 1$ (pour l'AG pur) et avec $\overline{T} = 3$ (pour l'AG intégré au sein d'une boucle *SBO*) réalisent des résultats semblables à Minamo, voire même de meilleurs résultats, tout en divisant le temps d'exécution par deux. Nous garderons donc ces deux versions pour le Chapitre 7, où nous comparerons les méthodes intéressantes testées.



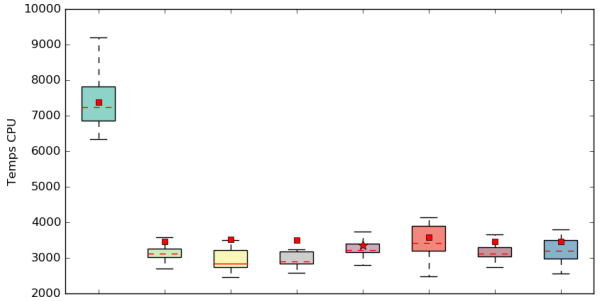
(a) *Rastrigin 10D* - Valeurs finales de la fonction objectif



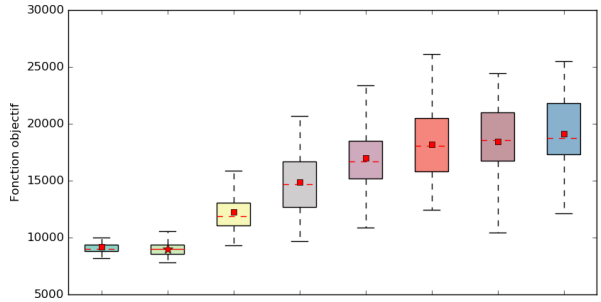
(b) *Rastrigin 10D* - Temps d'exécution



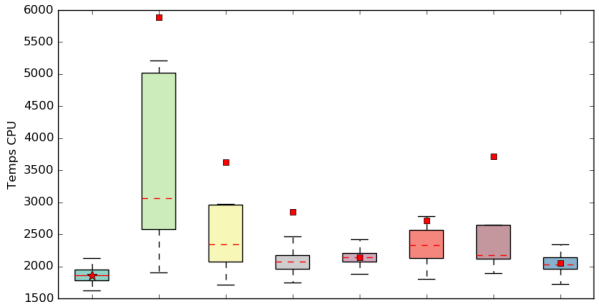
(c) *G7* - Valeurs finales de la fonction objectif



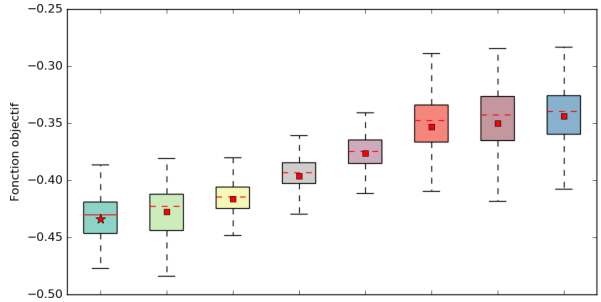
(d) *G7* - Temps d'exécution



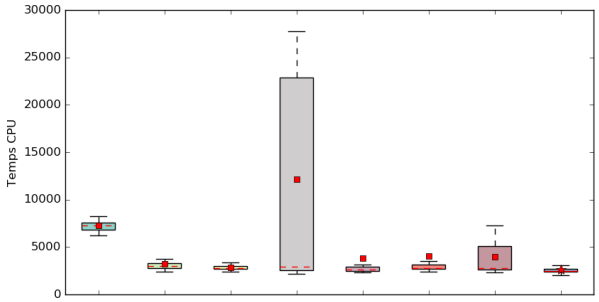
(e) *G10* - Valeurs finales de la fonction objectif



(f) *G10* - Temps d'exécution

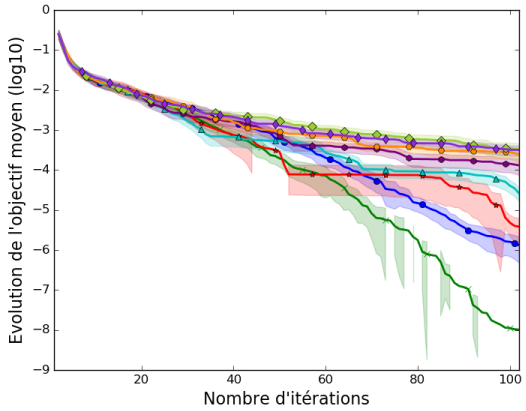


(g) *G2 plog 10* - Valeurs finales de la fonction objectif

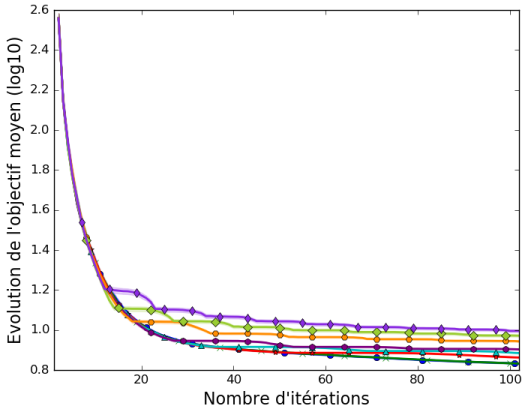


(h) *G2 plog 10* - Temps d'exécution

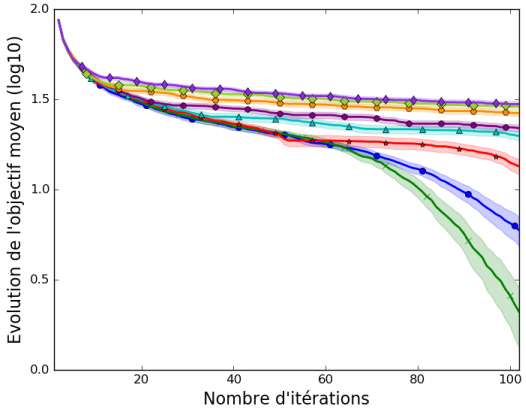
FIGURE 4.3 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *Saw-Tooth* et Minamo sur les cinq cas tests restants avec l'AG pur. Celles de gauche concernent la fonction objectif, tandis que celles de droite concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.



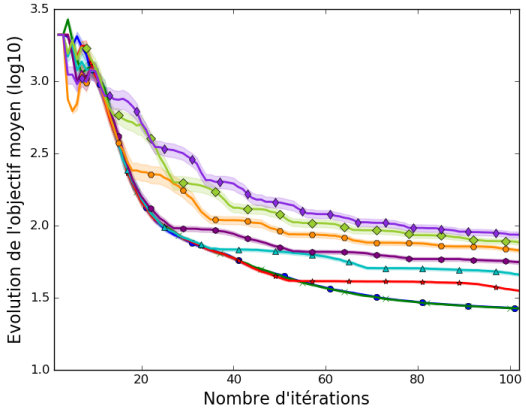
(a) *Rosenbrock 2D* - Graphe de convergence



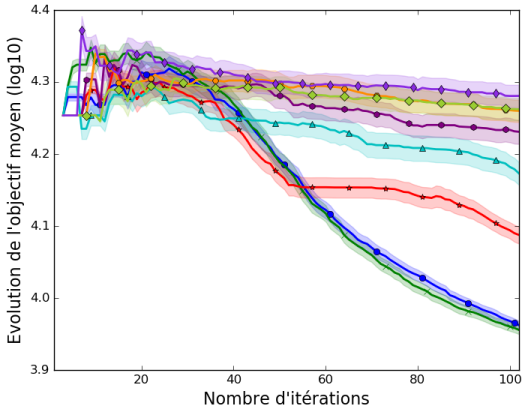
(b) *Rosenbrock 10D* - Graphe de convergence



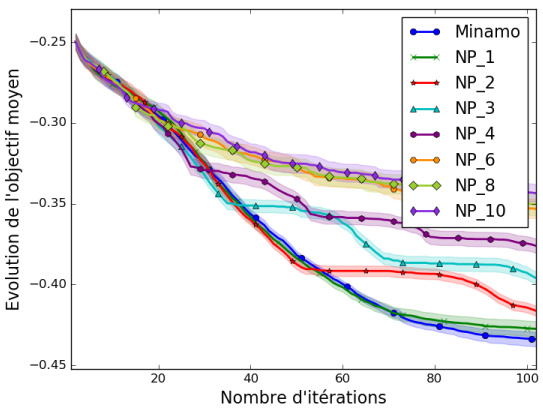
(c) *Rastrigin 10D* - Graphe de convergence



(d) *G7* - Graphe de convergence

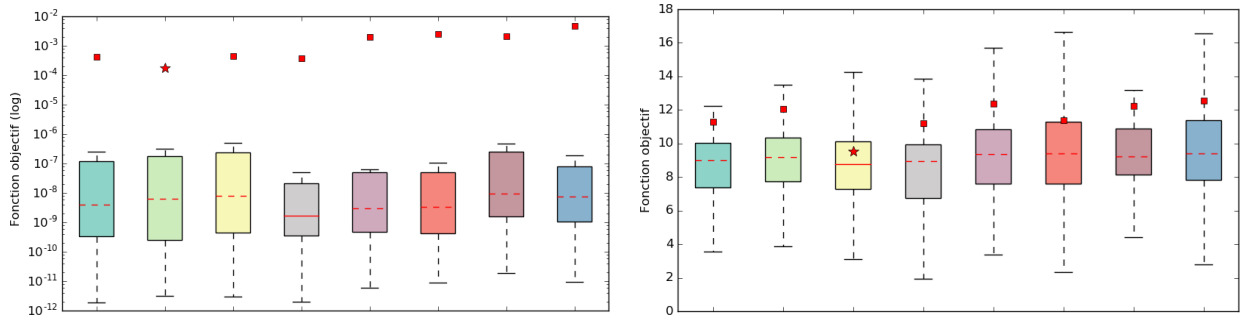


(e) *G10* - Graphe de convergence



(f) *G2 plog 10* - Graphe de convergence

FIGURE 4.4 – Ces graphes représentent la convergence moyenne pour les différentes versions testées de *Saw-Tooth* et Minamo sur les six cas tests avec l'AG pur.

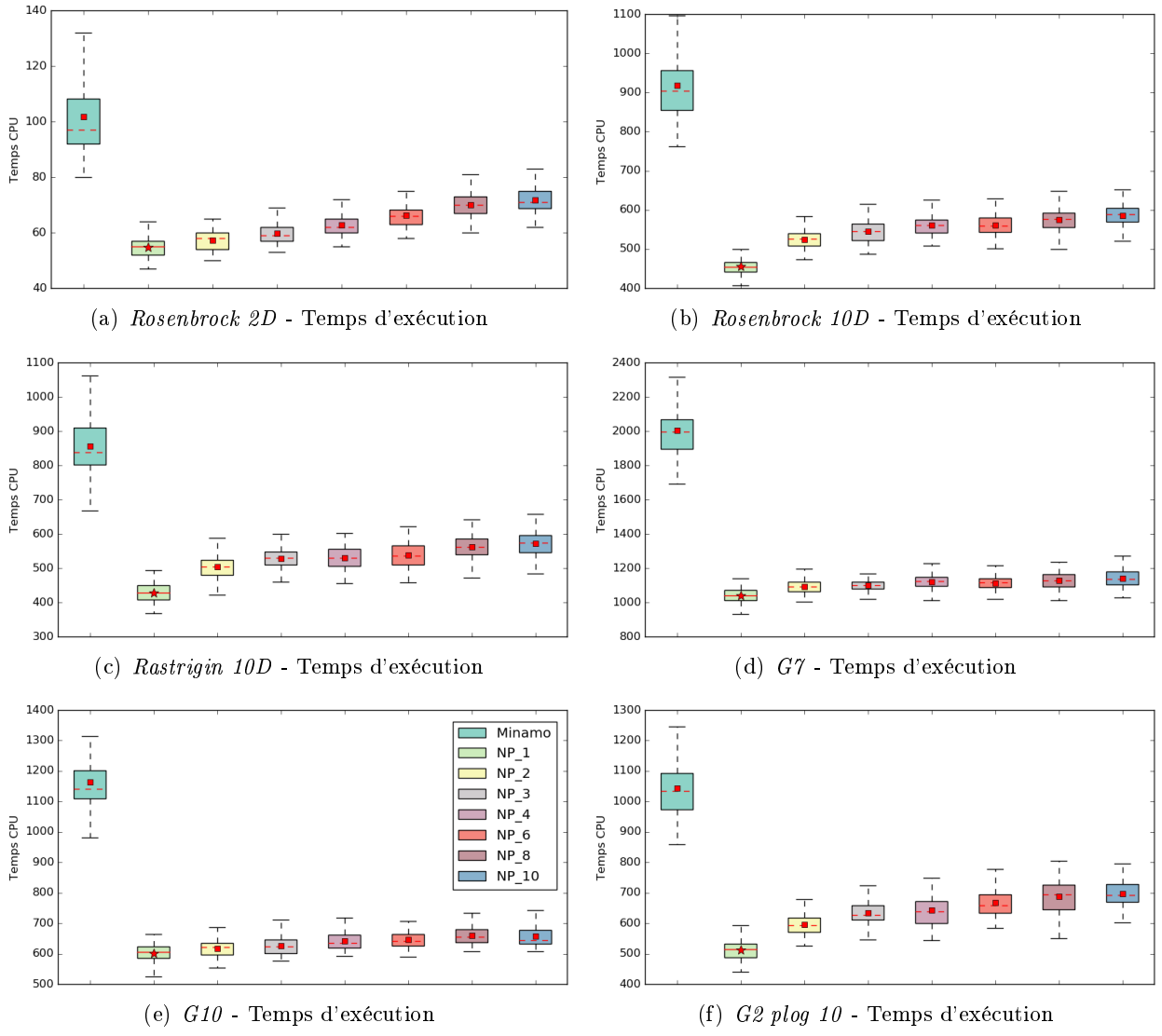


(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif (b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif (d) *G7* - Valeurs finales de la fonction objectif

(e) *G10* - Valeurs finales de la fonction objectif (f) *G2 plog 10* - Valeurs finales de la fonction objectif

FIGURE 4.5 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *Saw-Tooth* et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent les valeurs finales des fonctions objectifs. La légende est la même pour toutes ces figures.



(a) *Rosenbrock 2D* - Temps d'exécution

(b) *Rosenbrock 10D* - Temps d'exécution

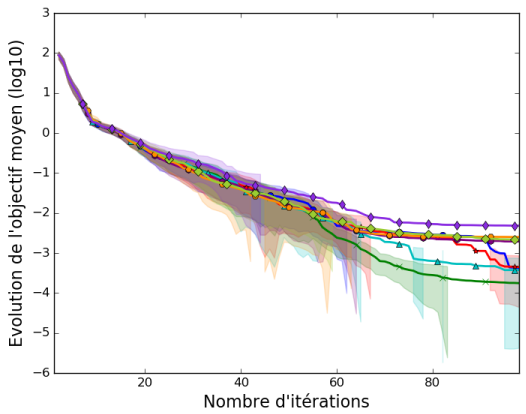
(c) *Rastrigin 10D* - Temps d'exécution

(d) *G7* - Temps d'exécution

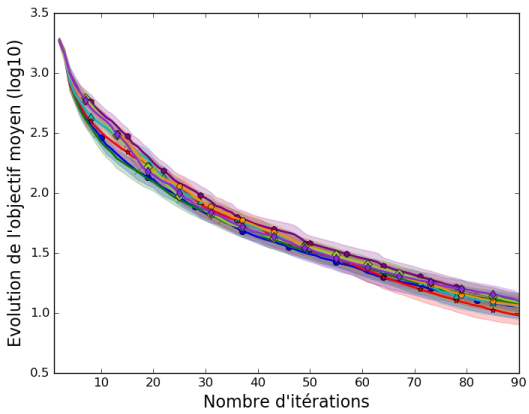
(e) *G10* - Temps d'exécution

(f) *G2 plog 10* - Temps d'exécution

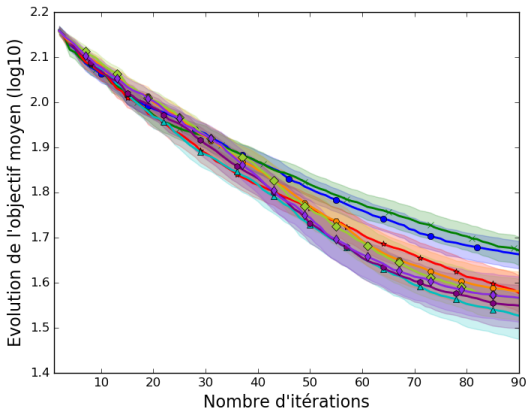
FIGURE 4.6 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *Saw-Tooth* et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.



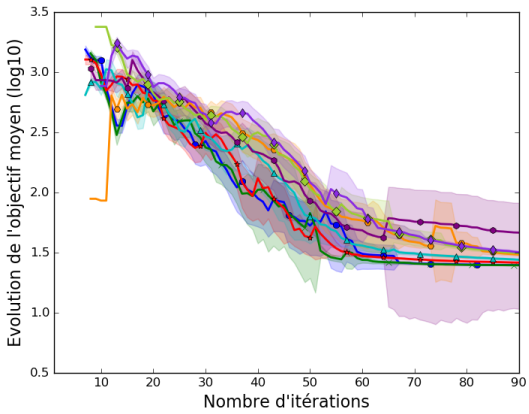
(a) *Rosenbrock 2D* - Graphe de convergence



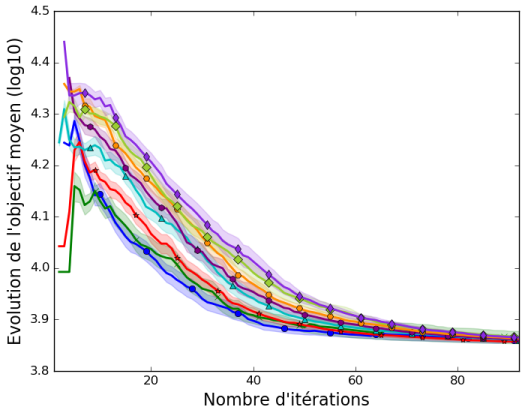
(b) *Rosenbrock 10D* - Graphe de convergence



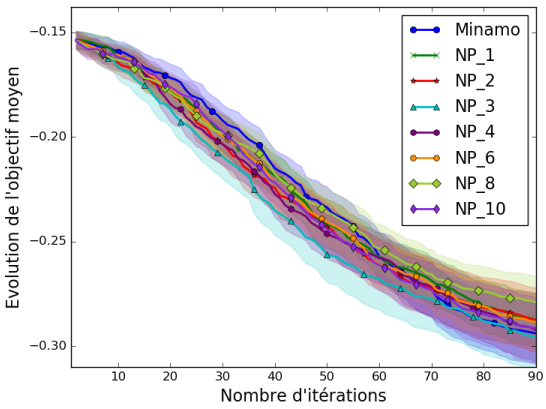
(c) *Rastrigin 10D* - Graphe de convergence



(d) *G7* - Valeurs finales de la fonction objectif

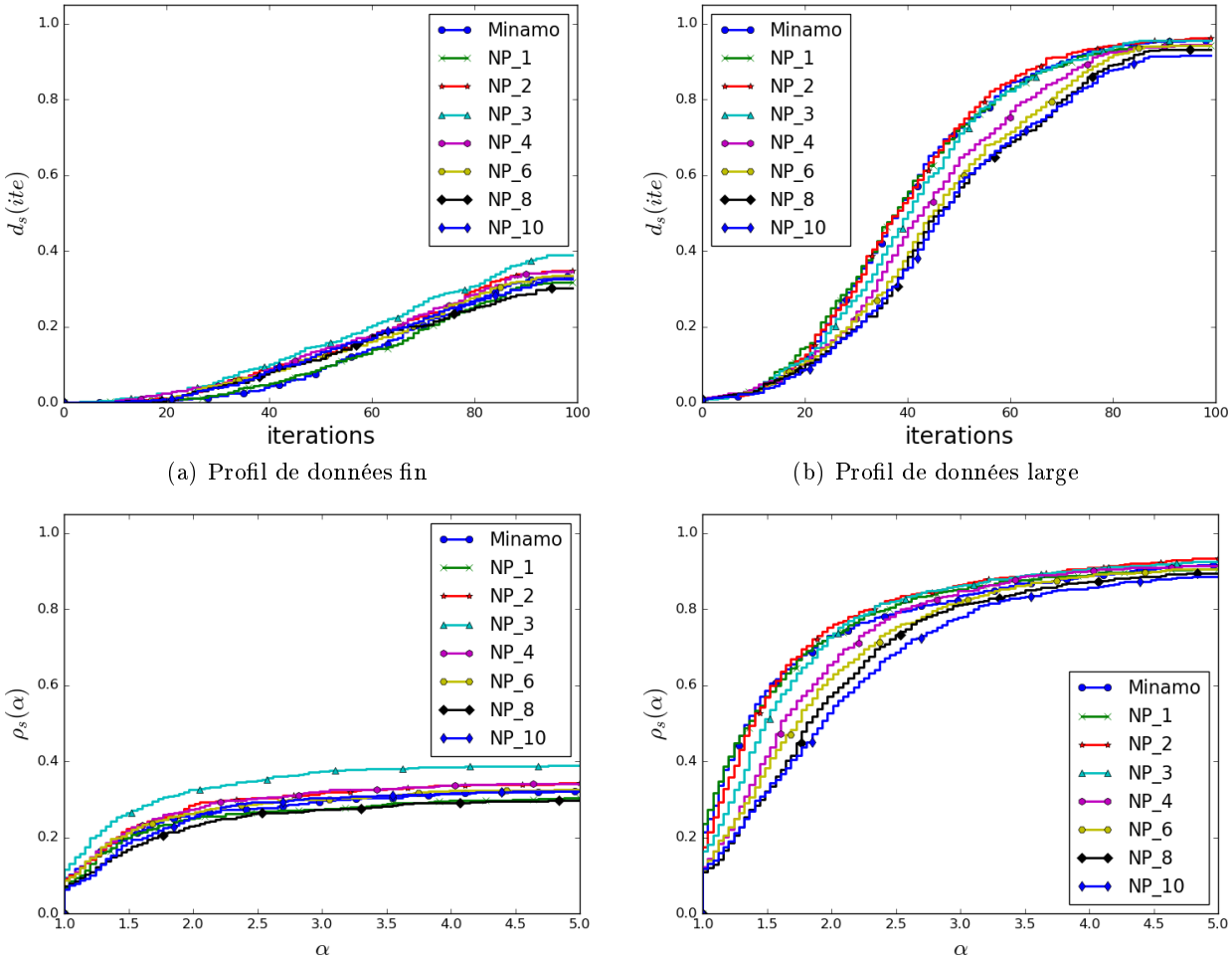


(e) *G10* - Graphe de convergence



(f) *G2 plog 10* - Graphe de convergence

FIGURE 4.7 – Ces graphes représentent la convergence moyenne pour les différentes versions testées de *Saw-Tooth* et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*.



(a) Profil de données fin

(b) Profil de données large

(c) Profil de performance fin

(d) Profil de performance large

FIGURE 4.8 – Les graphiques de profil de données et de performance sur les six cas tests pour les méthodes *Saw-Tooth* exécutées au sein de la boucle *SBO*.

Chapitre 5

GAVaPS - explication, implémentation et résultats

Dans ce chapitre, nous commencerons par expliquer la philosophie de la méthode introduite par Arabas et al. [2] (*GAVaPS*, signifiant *Genetic Algorithm with Varying Population Size*) pour régler le nombre d'individus au sein de l'AG. Ce réglage de la taille de la population est un contrôle adaptatif, car il tient compte de l'état de la population pour déterminer la taille de la population. Pour ce faire, ils introduisent à la naissance une espérance de vie pour chaque point créé. Ceci permet de garder de bons individus durant plusieurs générations et de moins bons durant quelques itérations seulement afin de garantir de sortir de minima locaux. Cette méthode originale introduit deux paramètres. Ainsi, après avoir réalisé une section décrivant l'implémentation réalisée au sein de Minamo, nous réglerons ces deux paramètres sur les six cas tests décrits dans la Section 3.4 pour l'AG pur et pour l'AG dans une boucle *SBO*. Nous espérons toujours qu'une amélioration en termes de temps sera observée, sans détérioration de la convergence vers l'optimum. Pour finir, nous résumerons les résultats et citerons une piste d'amélioration pour l'implémentation.

5.1 Méthodes originales

Arabas et al. [2] partent du constat suivant : la pression de sélection (*SP*, pour *Selective Pressure* en anglais) ne doit pas être trop élevée, auquel cas la population ne serait pas assez diversifiée. De plus, si la *SP* est trop faible alors la recherche de solution devient inefficace car celle-ci reviendrait à une recherche aléatoire. Ainsi, il faut sélectionner de bons individus au cours des itérations pour les générations suivantes, mais garder de moins bons qui peuvent alors garantir une certaine diversification dans les gènes. Pour ce faire, ils introduisent le concept d'espérance de vie pour chaque individu, qui dépend de sa valeur de fitness et des valeurs de fitness des autres individus, afin de déterminer le nombre d'itérations que chaque individu doit rester dans la population. Pour y parvenir, ils retiennent pour chaque individu le nombre d'itérations durant lesquels ils ont déjà été présent. Ceci est l'âge des individus. A côté de cela, nous savons qu'il est préférable de faire varier le paramètre μ au cours des itérations, comme l'a déjà montré le chapitre précédant (Chapitre 4). Cette variation de la taille de la population est alors déterminée par ces espérances de vie et ceci devrait être aussi bénéfique. Notons que cette méthode remplace la sélection d'individus pour la génération suivante et qu'elle va introduire les enfants dans la génération courante, sans la remplacer. Ainsi toute sélection par tournoi, ou autre, après la génération d'enfants devra être enlevée. Arabas et al. [2] proposent l'Algorithme 2 repris ici.

Comme nous l'avons dit, Arabas et al. [2] ajoutent une espérance de vie $EV(i)$ pour chaque individu i. Un enfant, une fois évalué par la fonction de fitness, reçoit une espérance de vie qui déterminera le nombre de générations qu'il pourra survivre. Dans l'article, ils stockent une valeur supplémentaire pour chaque individu qui concerne l'âge. Pour un individu quelconque, celle-ci com-

Algorithme 2 : GAVaPS - original [2]

```
début
    t = 0 ;
    initialiser P(t) ;
    évaluer P(t) ;
    donner une espérance de vie à tous les individus de P(t) ;
    tant que la condition de fin n'est pas atteinte faire
        augmenter l'âge de tous les individus d'une unité ;
        croiser et générer un certain nombre d'enfants E(t) ;
        muter les enfants E(t) ;
        évaluer les enfants E(t) ;
        donner un âge aux enfants E(t) ;
        P(t+1) ← P(T) ∪ E(t) ;
        pour tous les individus i de P(t) qui ont atteint leur espérance de vie faire
            enlever ces individus i de P(t) ;
        t ← t + 1 ;
```

mence de zéro et fini à l'espérance de vie, en étant augmenté d'une unité à chaque itération. Une fois que l'individu est aussi vieux que sont espérance de vie, alors celui-ci est enlevé de la population. L'espérance de vie $EV(i)$ pour chaque individu i est alors calculée une unique fois et ne devrait pas être modifiée sauf, nous l'imaginons bien, dans des cas extrêmes où la population ne contient plus d'individus. Il existe trois manières de la calculer selon eux.

1. L'allocation **proportionnelle** tient compte de la valeur de fitness moyenne fit_{moy}, de la valeur de fitness $fit(i)$ de l'enfant i, d'une espérance de vie minimale EV_{min} et d'une espérance de vie maximale EV_{max}.
2. L'allocation **linéaire** utilise EV_{min}, EV_{max}, la plus petite valeur de fitness déjà calculée lors de l'entièreté des itérations (qui précèdent l'itération courante) $fit_{min,abs}$ et la plus grande $fit_{max,abs}$, ainsi que $fit(i)$.
3. L'allocation **bi-linéaire** est le calcul le plus complexe et nécessite de connaître la valeur de fitness minimale fit_{min} et maximale fit_{max} de la population courante, ainsi que fit_{moy}, $fit(i)$, EV_{min} et EV_{max}.

Chaque approche permet de mettre à l'échelle, entre EV_{min} et EV_{max}, l'espérance de vie souhaitée selon l'état actuel (voire également passé) de la recherche. Ces deux bornes sont alors deux nouveaux paramètres que la méthode introduit. Notons qu'Arabas et al. [2] fixent EV_{min} à 1 et EV_{max} à 7 pour des problèmes de maximisation, tandis que Bäck, Eiben et al. [23] déterminent EV_{max} à 11 (car des exécutions préliminaires leur permettaient d'obtenir de meilleures performances), en gardant EV_{min} à 1, dans la formule bi-linéaire, pour des problèmes de minimisation. Il est alors évident que nous devrons déterminer quelle valeur pour EV_{max} correspondra le mieux dans Minamo et que nous garderons également EV_{min} à 1. Les formules de ces allocations concernent les problèmes de maximisation dans l'article d'Arabas et al. [2]. Il faut alors les traduire pour des problèmes de minimisation. Selon le type de méthodologie choisie avant le lancement de l'AG, l'espérance de vie $EV(i)$ pour l'individu i est alors calculée comme suit :

— pour l'allocation proportionnelle (PROP)

$$EV(i) = \min \begin{cases} EV_{max} \\ EV_{min} + \varphi \left(\dfrac{fit_{max} - fit(i)}{fit_{moy}} \right) \end{cases} \tag{5.1}$$

— pour l'allocation linéaire (LIN)

$$EV(i) = EV_{max} - 2\varphi\left(\frac{fit(i) - fit_{min,abs}}{fit_{max,abs} - fit_{min,abs}}\right) \tag{5.2}$$

— et pour l'allocation bi-linéaire (BILIN)

$$EV(i) = \begin{cases} EV_{min} + \varphi\left(\dfrac{fit_{max} - fit(i)}{fit_{max} - fit_{moy}}\right) & \text{si } fit(i) \geq fit_{moy} \\ \dfrac{1}{2}(EV_{max} + EV_{min}) + \varphi\left(\dfrac{fit_{moy} - fit(i)}{fit_{moy} - fit_{min}}\right) & \text{sinon.} \end{cases} \tag{5.3}$$

où $\varphi = \frac{1}{2}(EV_{max} - EV_{min})$.

Philosophiquement, plus un individu a sa valeur de fitness petite, plus celui-ci doit vivre longtemps. Dans le Tableau 5.1 contenant dix individus fictifs possédant chacun une valeur de fitness, nous illustrons le calcul de l'espérance de vie pour chacune des formules, en fixant EV_{min} à 1 et EV_{max} à 7. Nous constatons que la première formule (proportionnelle) favorise considérablement les individus qui sont (bien) meilleurs, ou aussi bons, que la moyenne des valeurs de fitness fit_{moy} (valant 3.8722 dans cet exemple). Par exemple, les individus qui ont une valeur de fitness comprise entre le minimum (0.10) et la moyenne, ont une espérance de vie de sept (voire de six) générations ; tandis que les deux individus les plus médiocres (ayant leur valeur de fitness égale à 9.00 ou à 10, dans cet exemple) ont une espérance de vie que d'une ou deux itération(s). Or, les deux autres formules (linéaire et bi-linéaire) définissent de manière plus "douce", de façon mieux répartie, l'espérance de vie entre ces individus fictifs. La formule bi-linéaire garantit, en plus, que l'individu qui aurait sa valeur de fitness égale à fit_{moy} posséderait comme espérance de vie, l'espérance de vie moyenne ($\frac{EV_{max}+EV_{min}}{2}$, qui est ϕ).

individu i	$fit(i)$	PROP	LIN	BI-LIN
1	0.10	7.00	7.00	7.00
2	0.25	7.00	6.91	6.88
3	0.50	7.00	6.76	6.68
4	1.00	7.00	6.45	6.28
5	2.00	7.00	5.85	5.49
6	3.00	6.42	5.24	4.69
7	4.00	5.65	4.64	3.94
8	5.00	4.87	4.03	3.45
9	9.00	1.77	1.61	1.49
10	10.0	1.00	1.00	1.00

TABLE 5.1 – Ce tableau illustre pour dix individus fictifs le calcul de l'espérance de vie selon les différentes formules.

Un autre paramètre est le taux de reproduction, noté ρ, qui ne doit pas être confondu avec le taux de croisement p_c. Le taux de reproduction ρ détermine le nombre d'enfants qui seront créés lors de la reproduction, à l'inverse du nombre de parents qui est déterminé par p_c. Ce pourcentage permet à la population de ne pas s'accroître de manière excessive, selon cette article d'Arabas et al. [2]. En effet, si de bons individus sont trouvés pendant plusieurs itérations, alors les enfants vont obtenir une bonne espérance de vie, puisque les anciens individus seront devenus moins bons. Ainsi, pour empêcher d'avoir de plus en plus d'enfants, il est nécessaire d'en créer moins que le nombre d'individus présents dans la population. De cette manière, à l'itération t, le nombre d'enfants λ_t est déterminé selon le nombre d'individus présents dans la population (μ_t) de la manière suivante :

$$\lambda_t = \rho \times \mu_t.$$

Également, la taille de la population à l'itération suivante $\mu(t+1)$ est la taille de la population à l'itération courante $\mu(t)$ augmentée du nombre d'enfants générés et où est retranché le nombre d'individus qui décèdent. Celle-ci est calculée comme suit :

$$\mid [P(t) \cup E(t)](age = EV) \mid,$$

où $\mid . \mid$ est la cardinalité, car ils ont atteint leur espérance de vie. Mathématiquement, cela se traduit par la formule suivante :

$$\mu(t+1) = \mu(t) + \lambda(t) - \mid [P(t) \cup E(t)](age = EV) \mid .$$

Notons que, d'un point de vue algorithmique, p_c est différent de ρ mais, d'un point de vue philosophique dans certains cas, ils peuvent être semblables. En effet, si nous considérons la taille de la population inchangée durant les itérations et si deux individus sont créés lors du croisement de deux parents, alors le nombre de parents utilisés lors de cette étape est en moyenne $100 \times \rho \times \mu$, ce qui est égal au taux de croisement.

Concernant la taille de la population initiale μ_0, cette approche est la seule méthode qui ne commence pas avec une taille proportionelle au nombre d'individus ($100 \times n_D$, où n_D est le nombre de dimension). En effet, dans l'article d'Arabas et al. [2] $\mu_0 = 20$ tandis que, dans l'article de Bäck, Eiben et al. [23], $\mu_0 = 60$. Une taille initiale petite est justifiée par le fait suivant : au début des itérations, la population va croître rapidement afin de découvrir une zone intéressante dans l'espace de représentation.

Dans l'article d'Arabas et al. [2], après avoir fixé μ_o à 20, EV_{min} à 1 et EV_{max} à 7, ils ont réalisé une analyse sur le paramètre ρ. Ils ont conclu que, pour les trois stratégies, la valeur optimale était approximativement 0.4. Après cela, ils ont réalisé une analyse concernant les trois variantes en fixant ρ à 0.4. Ils en ont conclu que la stratégie linéaire est la plus performante mais la plus coûteuse également (en terme du nombre de fois que la fonction est évaluée). Par contre, la stratégie bi-linéaire est la moins coûteuse mais elle n'est pas aussi bonne que la linéaire. Pour finir, l'allocation proportionnelle fournirait des résultats moyens avec un coût moyen.

Dans la section suivante, nous allons présenter l'implémentation de cette stratégie que nous avons réalisée dans Minamo. Nous avons dû adapter Minamo pour accueillir cette approche, en réalisant plusieurs adaptations. Ainsi, nous jugeons qu'il est préférable de déterminer la stratégie et les paramètres les plus adéquats dans une section ultérieure.

5.2 Implémentation et adaptations dans Minamo

Comme l'implémentation a été réalisée dans Minamo qui est un programme déjà existant, il a fallu réaliser quelques adaptations dans son implémentation. Par exemple, nous avons été contraints de gérer une population supplémentaire qui accueille tous les individus qui ont un âge positif car, dans l'implémentation qui existait, la population était d'office supprimée à chaque itération lors des différents processus. Dans cette section, nous allons décrire ces différentes modifications réalisées.

Fusion de l'espérance de vie et de l'âge

Nous n'avons pas introduit pour chaque individu une espérance de vie et un âge. Pour un individu i, nous lui attribuons une unique valeur qui est $EV(i)$, représentant l'espérance de vie restante. Celle-ci est alors diminuée à chaque itération et correspond au nombre d'itérations restantes avant qu'il décède. Une fois $EV(i) \leq 0$, l'individu i est enlevé de la population.

Gestion de la population

Dans la conception de cette implémentation, nous nous sommes questionnés sur la manière de garder des individus de génération en génération ; comme dans le code existant, les enfants remplacent les individus présents dans la population. En d'autres termes, $P(t) = E(t-1)$, à l'itération $t > 0$, dans Minamo. Dans Minamo, il n'était pas possible d'ajouter les enfants dans la population courante, car sinon l'architecture de l'implémentation de Minamo devrait être repensée. Nous avons alors préféré créer une population de survie qui contient tous les individus vivants. Celle-ci sera utilisée pour créer les enfants et sera gérée à un unique emplacement, lors de la gestion de la méthode *GAVaPS*. Notons que d'un point de vue algorithmique dans Minamo, celle-ci correspond à une population supplémentaire (une population de survie, notée P'). Cependant cette population doit correspondre en réalité à la population courante dans la méthodologie de l'AG. En effet, dans l'AG, nous pouvons sélectionner des enfants pour les introduire dans la population courante, nous pouvons remplacer cette population par les enfants, nous pouvons sélectionner des individus parmi les parents et les enfants pour générer la population courante, etc. En résumé, la population de survie P' correspond à la population P du véritable algorithme génétique théorique mais pas dans ce chapitre.

A la première itération de l'AG, il faut initialiser la population de survie $P'(0)$ en reprenant tous les individus de la population initiale $P(0)$. Ici, la taille de $P'(0)$ (notée $\mu'(0)$) est de la même taille que $P(0)$ ($\mu(0)$), ce qui ne sera plus le cas dans les prochaines itérations. Ensuite, l'attribution de l'espérance de vie pour tous les individus de $P'(0)$ est réalisée selon une des trois formulations définies en section précédente (proportionnelle, linéaire ou bi-linéaire). Puis, le croisement, la mutation et l'évaluation peuvent être réalisés.

A l'itération suivante, la diminution d'une unité de l'âge des individus présents dans $P'(0)$ est réalisée. Après cela, chaque individu i de $P'(0)$, où $EV(i) \leq 0$, se voit supprimé. Ensuite, les enfants générés $E(0)$ à l'itération précédente sont rajoutés à la fin de la population de survie pour former $P'(1)$. Puis, les espérances de vie pour les derniers individus ajoutés sont calculées par la même stratégie qu'à l'itération 0.

Notons que, lors de quelques exécutions préliminaires, nous avions eu une population de survie qui ne contenait qu'un individu, après seulement quelques itérations. Ceci ne permettant pas de réaliser de croisement, la population de survie était devenue vide après quelques nouvelles itérations. Ces exemples nous ont convaincu qu'il fallait introduire un nombre minimal d'individus μ_{min} pour la population de survie. Celui-ci est déterminée comme suit :

$$\mu_{min} = \begin{cases} \mu(0), & \text{si } 5 \times n_D > \mu(0) \\ 5 \times n_D, & \text{sinon} \end{cases}$$

où n_D est le nombre de dimension du problème considéré. Ce calcul est un compromis entre la valeur initiale $\mu(0)$ et le calcul de μ_{min} déjà introduit dans le Chapitre 4 qui, pour un problème en 2D, fixait μ_{min} à 10. Ainsi, si le nombre d'individus qui devaient être supprimés implique que la population de survie contienne moins d'individus que μ_{min}, alors la suppression n'était pas réalisée. Par ce fait, nous étions obligés de sauvegarder les indices des individus qui devaient être supprimés dans un ensemble noté I_{sup}, dans ce mémoire. La suppression est donc réalisée si

$$\mu'(0) - \mid I_{sup} \mid > \mu_{min},$$

où $\mid . \mid$ est le calcul de la cardinalité. Par contre, aucune condition sur le nombre maximal d'individus n'a été introduite.

De manière générale, à l'itération $t > 0$, la gestion préliminaire de la population de survie suit donc les étapes suivantes :

1. $\forall i \in P'(t-1)$, $EV(i) \leftarrow EV(i) - 1$, c'est-à-dire que l'espérance de vie de tous les individus est réduite ;
2. créer I_{sup} où $\forall i \in I_{sup}$, $EV(i) \leq 0$, c'est-à-dire que I_{sup} contient tous les index des individus qui doivent être supprimés ;
3. si $\mu'(t-1) - \mid I_{sup} \mid > \mu_{min}$, alors supprimer de $P'(t-1)$ les individus i de l'ensemble I_{sup}, sinon ne rien faire ;
4. supprimer I_{sup} ;
5. $P'(t) \leftarrow P'(t-1) \cup E(t-1)$, c'est-à-dire que les enfants sont ajoutés à la fin de la population de survie (réduite) ;
6. calculer l'espérance de vie de chaque individu dernièrement ajouté selon une des trois formulations de GAVaPS.

Stockage de l'EV (dans une map) et fonction de hachage

Sous les conseils de l'équipe Minamo, l'espérance de vie ne doit pas être stockée dans les individus, car elle ne serait pas souvent utilisée. En effet, celle-ci n'est utile que si l'utilisateur spécifie vouloir travailler avec GAVaPS. Ainsi, nous avons créé un dictionnaire, également appelé map, qui contient comme clés les "identifiants" associés aux individus et comme valeurs les espérances de vie.

Initialement, nous avions utilisé les identifiants automatiquement générés par Minamo pour les individus. Cependant, Minamo n'en génère que pour les individus de l'AG pur. Ainsi, quand nous sommes passés à des exécutions avec l'AG intégré dans une boucle *SBO*, les identifiants n'étaient plus générés. Une solution, qui aurait été rapide à implémenter mais qui aurait pu être lourde dans certains cas, nous a été déconseillée par l'équipe Minamo. Celle-ci aurait été de générer des identifiants pour tous les individus créés, dans l'AG de la boucle *SBO*. La solution proposée par l'équipe Minamo est d'utiliser une fonction de hachage sur les exons comme identifiant.

Une fonction de hachage est une fonction qui synthétise une information en une autre information plus concise, en utilisant des opérations mathématiques et informatiques. Dans notre cas, les exons des individus sont synthétisés en un entier. Ainsi, un individu i_1, qui est différent d'un individu i_2 par ses chromosomes, va probablement recevoir un entier différent de l'individu i_2 suite à la fonction de hachage. Si deux individus obtiennent la même valeur par la fonction de hachage (valeur de *hash*), alors une collision se produit. Le risque de collision est moindre. De plus, pour deux individus possédant des chromosomes proches, la fonction de hachage garantit des valeurs différentes.

Notons que, par l'élitisme présent dans l'AG de Minamo, certains individus peuvent être des copies (avec une légère modification si la mutation a été réalisée sur les chromosomes de ces individus). Cependant, des copies exactes d'individus de la population de survie ne doivent pas être introduites dans celle-ci lors d'itérations, si aucune mutation n'a été réalisée. En effet, admettons qu'un individu i_E présent dans les enfants $E(t)$ soit identique à un individu $i_{P'}$ de $P'(t)$ à l'itération $t+1$, si nous introduisons i_E dans $P'(t)$ alors l'espérance de vie de $i_{P'}$ sera indéniablement modifiée, ce qui est interdit. Pour ce faire, lors de la copie des enfants $E(t)$ dans $P'(t)$ (réduit) pour donner $P'(t+1)$, il faut vérifier que ces individus sont différents, par la fonction de hachage.

Dans le cas d'une collision d'un enfant i_E de $E(t)$ à ajouter dans $P'(t)$ avec un individu $i_{P'}$ de $P'(t)$ à l'itération $t+1$, il faut la traiter. Donc, dans le cas où ils ne sont pas identiques et que, par malchance, ils ont la même valeur de *hash*, le mieux à faire est de garder l'individu qui a la meilleure valeur de fitness. Ainsi, trois cas sont possibles pour i_E et $i_{P'}$.

1. Si $fit(i_E) < fit(i_{P'})$, alors il faut supprimer l'espérance de vie de $i_{P'}$ de la map et retirer l'individu $i_{P'}$ dans $P'(t)$. Ensuite, l'enfant peut être ajouté dans la population de survie et le calcul d'espérance de vie devra être réalisé une fois l'entièreté des enfants ajoutés.

2. Si $fit(i_E) > fit(i_{P'})$, alors il ne faut pas ajouter l'enfant i_E dans la population de survie. De plus, la map et la population de survie gardent les informations concernant $i_{P'}$.
3. Si $fit(i_E) = fit(i_{P'})$, alors il est probable que l'individu i_E possède les mêmes chromosomes que $i_{P'}$. Ainsi, par facilité nous gardons $i_{P'}$ dans la population de survie. Ceci n'a pas d'impact sur la convergence. En effet, le meilleur individu de la population de survie, par élitisme, sera recréé à chaque itération dans les enfants (i_E) et, lorsqu'il décédera, i_E sera directement rajouté dans la population de survie.

Découverte que fit_{max} ne diminuait pas

Un problème que nous avions eu pour *GAVaPS* avec l'allocation bi-linaire (BILIN) est que la population (appelée population de survie dans la partie précédente) croissait énormément durant les itérations, pour finir avec un nombre d'individus créé durant les itérations bien plus haut que le nombre d'individus créés durant les itérations de Minamo (où la taille était fixe). Pour comprendre ce qui pouvait être la cause nous avions sauvegardé dans un fichier l'évolution de la recherche au cours des itérations, en retenant le numéro de l'itération, le nombre d'individus présents dans la population, le nombre total d'individus créés durant les itérations précédentes, la valeur de fitness minimale de la population fit_{min} et la valeur maximale fit_{max}. Nous avons alors découvert que la valeur de fit_{max} ne décroissait pas réellement, tandis que fit_{min} diminuait bel-et-bien. Ainsi, les formulations de l'allocation de l'espérance de vie qui considèrent fit_{moy} (qui ne diminue pas non plus) imposait aux individus proches (c'est-à-dire qui ont une valeur de fitness plus basse que fit_{moy}) de meilleures espérances de vie.

Pour mieux comprendre, prenons un cas fictif dans le Tableau 5.2, où à l'itération t (avec $t > 0$) nous avons la population issue de la génération précédente $P(t-1)$ et les enfants créés à l'itération précédente $E(t-1)$. Le moins bon individu de $P(t-1)$ a sa valeur de fitness égale à 1.5 ($fit_{max}(t-1) = 1.5$), tandis que des enfants dans $E(t-1)$ ont leur valeur de fitness plus grandes que 1.5 (par exemple, deux enfants ont leur valeur de fitness égale à 7.0 et 8.0) et nous n'avons pas amélioré fit_{min}. Dans ce cas, si nous ajoutons tous les enfants dans la population (formant ainsi $P(t)$) alors $fit_{max}(t) > fit_{max}(t-1)$ et il en va de même pour la valeur moyenne ($fit_{moy}(t) > fit_{moy}(t-1)$). Ainsi, plus nombreux seront les individus qui seront en dessous de la valeur de fitness moyenne et auront une meilleure espérance de vie. Ce souci se produit souvent durant l'exécution : par exemple, quand bon nombre d'individus convergent correctement tandis que quelques-uns sont à la traîne. Deux cas sont présentés dans le Tableau 5.2 en utilisant la formule de l'allocation bi-linéaire : soit nous ajoutons tous les enfants, soit nous n'ajoutons que ceux qui ont une valeur de fitness plus basse que $fit_{max}(t-1)$. Nous constatons qu'évidement nous ajoutons plus d'individus s'il n'y a pas de condition sur la décroissance de fit_{max}. De plus, les individus ajoutés lorsque nous n'imposons pas la décroissance de fit_{max} possèdent une espérance de vie plus élevée que si nous avions imposé cette décroissance (par exemple, l'enfant ayant sa valeur de fitness égale à 1.20).

Ainsi, pour régler ce désagrément, il faut retirer tous les enfants qui détériorent la décroissance de fit_{max}. D'un point de vue algorithmique, à l'itération $t > 0$, il faut vérifier pour tout $i_E \in E(t-1)$ que $fit(i_E) < fit_{max}(t-1)$ et retirer les individus i_E de $E(t-1)$ tel que $fit(i_E) \geq fit_{max}(t-1)$. Notons que cette étape est inutile dans le cas de l'allocation proportionnelle et linéaire, selon plusieurs tests réalisés en amont.

Définition de l'AG utilisé

Après avoir réalisé ces modifications, nous pouvons décrire cette méthode implémentée dans Minamo par l'Algorithme 3. Nous constatons à premier abord que cette méthode demandera beaucoup de temps de calcul pour gérer la population. Pour vérifier notre intuition, nous allons comparer

$fit(i_P)$ où $i_P \in P(t-1)$	$fit(i_E)$ où $i_E \in E(t-1)$
0.10	0.11
0.12	0.17
0.15	0.25
0.17	1.2
0.20	7.00
0.25	8.00
0.26	
0.90	
1.00	
1.50	

$fit(i)$ où $i \in E(t-1)$	$EV(i)$	$EV(i)$
0.11	6.99	6.95
0.17	6.91	6.66
0.25	6.80	6.26
1.20	**5.55**	**2.14**
7.00	1.53	/
8.00	1.00	/

TABLE 5.2 – Dans le tableau de gauche, nous donnons un exemple fictif d'une population d'individus $P(t-1)$ et d'une population d'enfants $E(t-1)$ où $\rho = 0.6$ qui doivent être fusionnées à l'itération t (où $t > 0$). Dans le tableau de droite, nous avons : à gauche, l'espérance de vie calculée selon BILIN en ajoutant tous les enfants et, à droite, l'espérance de vie calculée selon BILIN en n'ajoutant que les individus qui ne détériorent pas fit_{max}.

différentes valeurs pour les paramètres (EV_{max} et ρ) sur six cas tests, en regardant bien le nombre d'individus générés au total pour chaque calcul, le temps d'exécution et la solution obtenue. Des comparaisons avec Minamo seront également faites.

5.3 Résultats pour l'AG

Nous commencerons cette section par citer les seize combinaisons de paramètres pour l'implémentation de GAVaPS et nous argumenterons le choix de ceux-ci. Dans un même temps, nous expliquerons un choix que nous avons fait : nous avons exécuté toutes ces versions en utilisant la formulation bi-linaire (Équation 5.3) pour l'attribution de l'espérance de vie, avant de sélectionner quelques combinaison pour l'allocation proportionnelle (Équation 5.1) et linéaire (Équation 5.2). Ensuite, nous allons donc réaliser les comparaisons des résultats de l'AG pur contenant l'implémentation de GAVaPS, en deux temps : d'abord, les seize combinaison avec l'allocation bi-linaire, ensuite les deux autres allocations. Tous les résultats de chaque méthode présentés dans cette section ont été obtenus par trente exécutions différentes.

Seize combinaisons de paramètres

Rappelons que, dans l'article d'Arabas et al. [2], après avoir fixé la taille initiale de la population (μ_0) à 20, l'espérance de vie minimale (EV_{min}) à 1 et l'espérance de vie maximale (EV_{max}) à 7, ils ont réalisé une analyse sur le taux de reproduction ρ. Ils ont conclu que, pour les trois stratégies de calcul de l'espérance de vie pour chaque individu, qui sont proportionnelle (Équation 5.1), linéaire (Équation 5.2) et bi-linaire (Équation 5.3), la valeur optimale de ρ était approximativement 0.4.

Cependant, nous nous allons réaliser une analyse sur l'espérance de vie maximale (EV_{max}) et le taux de reproduction (ρ), où nous avons fixé l'espérance de vie minimale (EV_{min}) à 1 comme deux articles utilisent cette valeur (voir [2] et [23]). Nous avons alors décidé que EV_{max} pourra valoir 4, 5, 6 ou 7 et que ρ pourra valoir 0.3, 0.4, 0.5 ou 0.6. Le but est alors de savoir quelle combinaison de paramètres correspondra le mieux à *GAVaPS* implémenté dans Minamo.

Algorithme 3 : GAVaPS - implémenté dans Minamo

```
1  début
2      t = 0 ;
3      fixer ρ, EV_min, EV_max ;
4      calculer φ = ½(EV_max − EV_min) ;
5      créer un ensemble vide I_sup ;
6      initialiser P(t) (qui sera considéré comme les parents) ;
7      évaluer P(t) ;
8      donner une espérance de vie à tous les individus dans la map EV ;
9      tant que t < 100 faire
10         si t > 0 alors
11             pour tous les i_E ∈ E(t−1) tq fit(i_E) ≥ fit_max(t−1) faire
12                 ajouter i_E dans l'ensemble I_sup ;
13             supprimer de E(t−1) les individus ayant les indices dans I_sup ;
14             vider l'ensemble I_sup ;
15             diminuer d'une unité l'espérance de vie de tous les individus de P'(t−1) ;
16             pour tous les individus i_P' de P'(t) faire
17                 si EV(i_P') ≤ 0 alors
18                     ajouter i_P' dans l'ensemble I_sup ;
19             si μ'(t−1)− | I_sup | > μ_min alors
20                 supprimer de P'(t−1) les individus ayant les indices dans I_sup ;
21             vider l'ensemble I_sup ;
22             P'(t) ← P'(t−1) ∪ E(t−1) (les enfants sont ajoutés à la fin) ;
23             pour tous les enfants i ajoutés dans P'(t) faire
24                 si il existe un ancien individu j (du début) de P'(t), tel que i et j sont en
                    collision avec leur valeur de hash alors
25                     si fit(i) < fit(j) alors
26                         ajouter j dans l'ensemble I_sup ;
27                         supprimer de la map EV l'information concernant j ;
28                     sinon
29                         ajouter i dans l'ensemble I_sup ;
30             supprimer de P'(t) les individus ayant les indices dans I_sup ;
31             vider l'ensemble I_sup ;
32             calculer EV(i) aux individus i restants qui ont été ajoutés dans P'(t) ;
33             P(t) ← P'(t) ;
34         sinon
35             P'(0) ← P(0)
36         croiser P(t) et générer un certain nombre d'enfants E(t) (au nombre de ρ × μ(t)) ;
37         muter les enfants E(t) ;
38         évaluer les enfants E(t) ;
39         t ← t + 1 ;
```

Nous allons d'abord tester ces seize combinaisons en utilisant l'allocation bi-linéaire (BILIN) pour le calcul de l'espérance de vie. Il est préférable d'utiliser cette stratégie, plutôt que l'allocation linaire (LIN) ou proportionnelle (PROP), pour les raisons suivantes :

— la condition sur la valeur de fitness maximale (fit_{max}), qui impose une réduction de cette valeur au cours des itérations, ne permet pas une diminution (considérable) de la taille de la population avec LIN et PROP, tandis que BILIN le permet, ce qui induira un temps d'exécution moindre (nous l'espérons) ;
— Eiben et Bäck [23] ont également exécuté leur stratégie (auto-)adaptative en utilisant uniquement le calcul bi-linéaire ;
— Arabas et al. [2] conclut leur article en précisant que BILIN est le calcul le moins coûteux, celui qui nécessite le moins d'évaluations de fonctions, mais qui aurait de moins bons résultats.

Ensuite, nous allons exécuter quelques versions avec le calcul proportionnel et le calcul linéaire, pour vérifier que les résultats obtenus avec BILIN sont meilleurs que PROP ou LIN.

Comparaison des seize combinaisons pour l'allocation bi-linéaire

Le Tableau 5.3 reprend les tags des seize combinaisons de paramètres exécutées avec la formule bi-linéaire, qui donnera aux tags l'information BILIN, (Équation 5.3) sur six cas tests et sur trente exécutions différentes, dont nous présenterons les résultats dans cette section.

Tag	EV_{max}	ρ
BILIN_4_03	4	0.3
BILIN_4_04	4	0.4
BILIN_4_05	4	0.5
BILIN_4_06	4	0.6
BILIN_5_03	5	0.3
BILIN_5_04	5	0.4
BILIN_5_05	5	0.5
BILIN_5_06	5	0.6

Tag	EV_{max}	ρ
BILIN_6_03	6	0.3
BILIN_6_04	6	0.4
BILIN_6_05	6	0.5
BILIN_6_06	6	0.6
BILIN_7_03	7	0.3
BILIN_7_04	7	0.4
BILIN_7_05	7	0.5
BILIN_7_06	7	0.6

TABLE 5.3 – Ce tableau reprend les seize combinaisons de GAVaPS testées au sein de Minamo en utilisant l'allocation bi-linaire, selon les paramètres EV_{max} et ρ.

Dans la Figure 1 et la Figure 2 de l'Annexe A, nous avons repris les boîtes à moustache concernant la valeur finale des différentes solutions pour les différents problèmes d'optimisation considérés. Nous retenons BILIN_4_05, BILIN_4_06 et BILIN_5_06. En effet, la convergence est améliorée, par rapport à Minamo, pour *Rosenbrock 2D* (Figure 1(a)) avec ces trois versions et ce sont les versions qui améliorent le plus Minamo. De plus, sur le problème *Rastrigin 10D* (Figure 1(c)), les versions BILIN_4_06 et BILIN_5_06 permettent une amélioration tandis que BILIN_4_05 ne détériore pas la convergence finale. Concernant la convergence finale des différentes combinaisons de paramètres sur le problème *Rosenbrock 10D* (Figure 1(b)), nous avons que ces trois versions ne détériorent pas la solution finale. A côté de cela, la version BILIN_5_06 ne détériore pas la convergence finale pour le problème *G2 plog 10* (Figure 2(c)), dire que les deux autres versions (BILIN_4_05 et BILIN_4_06) détériorent la convergence, mais pas autant que les autres combinaisons de paramètres que nous ne retenons pas. Pour le problème *G7* (Figure 2(a)), ses solutions finales sont légèrement détériorées par rapport à Minamo, pour les versions BILIN_4_05 et BILIN_4_06. Pour finir, toutes les versions exécutées sur *G10* (Figure 2(b)) semblent analogues entre elles mais détériorent la solution par rapport à Minamo, avec cependant BILIN_5_06 qui ressort parmi celles-ci.

Concernant le temps d'exécution de toutes ces seize méthodes, celui-ci est bien souvent réduit par rapport à Minamo comme le montre la Figure 3 et la Figure 4 de l'Annexe A, sauf pour le problème sans contraintes *Rosenbrock 10D* (Figure 3(b)) et le problème avec contraintes *G10* (Figure 4(b))

pour lesquels la durée d'exécution est parfois plus élevée pour certaines versions et parfois améliorée pour d'autres. Sinon, les trois combinaisons retenues dans le paragraphe précédant (BILIN_4_05, BILIN_4_06 et BILIN_5_06) permettent donc une amélioration du temps d'exécution pour les cas tests *Rosenbrock 2D*, *G7*, *G2 plog 10* et *G10*. Cependant, pour le problème *Rosenbrock 10* (Figure 3(b)), seule la méthode BILIN_4_05 est plus rapide que Minamo, tandis que BILIN_4_06 et BILIN_5_06 sont plus lentes que Minamo. Il en est de même pour *G10* (Figure 4(b)).

Les Figures 5 et 6 de l'Annexe A recensent le nombre d'évaluations de fonctions de chacune de ces seize méthodes sur les six cas tests. Nous constatons que toutes les méthodes permettent une nette réduction du nombre d'évaluations de fonctions, par rapport au nombre d'évaluations de fonctions par Minamo. En effet, la diminution minimale est de la moitié. Cependant, certaines méthodes diminuaient plus le nombre d'évaluations de fonctions, comme par exemple BILIN_4_03. Toutefois, une telle réduction entraînait de mauvais résultats, comme nous l'avons conclu précédemment.

Une remarque doit être faite en observant le temps d'exécution et le nombre d'évaluations de fonctions : deux problèmes (*Rosenbrock 10D* et *G10*) ont été résolu en un temps identique à Minamo pour deux méthodes retenues (BILIN_4_05 et BILIN_4_06) ou même empiré pour BILIN_5_06, tandis le nombre d'évaluations a été divisé par deux ! Pourquoi ? Nous pensons que cela est causé par la gestion de cette méthode : beaucoup d'étapes doivent être faites pour simplement sélectionner de bons individus. De plus, certaines étapes peuvent être plus longues que d'autres. Citons par exemple la gestion de la map, qui contient l'espérance de vie de chaque individu. Or, nous savons que *Rosenbrock 10D* n'est pas une méthode qui est la plus coûteuse en temps de calcul. Ainsi, le temps gagné pour ne pas avoir évalué les points est alors perdu pour gérer cette méthode, malheureusement.

Ainsi, dans la sous-section suivante, nous présentons les résultats de *GAVaPS* avec le calcul proportionnel et linéaire pour l'espérance de vie seulement sur les trois combinaisons retenues (qui sont $(EV_{max}, \rho) = (4, 0.5)$, $(EV_{max}, \rho) = (4, 0.6)$ et $(EV_{max}, \rho) = (5, 0.5)$), mais avec des calculs différents pour l'allocation de l'espérance de vie.

Comparaison des combinaisons retenues avec les deux autres allocations

Les six versions testées, qui concernent l'allocation proportionnelle et l'allocation linéaire pour le calcul de l'espérance de vie, sont détaillées dans le Tableau 5.4. Dans cette section, nous allons également comparer les résultats des trois combinaisons retenues précédemment (BILIN_4_05, BILIN_4_06 et BILIN_5_06).

Tag	Calcul	EV_{max}	ρ
BILIN_4_05	bi-linéaire (Équation 5.3)	4	0.5
BILIN_4_06	bi-linéaire (Équation 5.3)	4	0.6
BILIN_5_06	bi-linéaire (Équation 5.3)	5	0.6
PROP_4_05	proportionnel (Équation 5.1)	4	0.5
PROP_4_06	proportionnel (Équation 5.1)	4	0.6
PROP_5_06	proportionnel (Équation 5.1)	5	0.6
LIN_4_05	linéaire (Équation 5.2)	4	0.5
LIN_4_06	linéaire (Équation 5.2)	4	0.6
LIN_5_06	linéaire (Équation 5.2)	5	0.6

TABLE 5.4 – Ce tableau reprend les trois combinaisons de GAVaPS testées au sein de Minamo en utilisant l'allocation linéaire, l'allocation proportionnelle ou bi-linéaire, selon les paramètres EV_{max} et ρ.

Dans les Figures 7 et 8 de l'Annexe A, nous présentons la convergence finale de ces neufs versions de *GAVaPS*, en comparaison avec Minamo. De plus, dans cette même annexe, nous présentons le temps d'exécution de ces différentes versions dans les Figures 9 et 10 de l'Annexe A. Nous allons les commenter.

Les versions qui utilisent l'allocation linéaire (LIN_4_05, LIN_4_06 et LIN_5_06) détériorent considérablement les résultats finaux par rapport à Minamo. Particulièrement, la résolution des problèmes *Rosenbrock 2D*, *Rastrigin 10D* et *G7* est détériorée par rapport à Minamo. Cependant, le problème *G2 plog 10* est résolu par ces versions de manière équivalente à Minamo, avec par contre un temps réduit (Figures 10(c)). Ceci ne représentant pas assez d'amélioration, nous ne retiendrons alors aucune de ces méthodes linéaires pour le Chapitre 7, où une comparaison entre les meilleures méthodes de ce mémoire sera réalisée.

Ensuite, pour les versions bi-linéaires et proportionnelles, nous retenons BILIN_4_05, BILIN_4_06, PROP_4_05 et PROP_4_05 pour la section suivante. Ce choix est réalisé par un compromis entre l'amélioration (voire de la non-régression) de la solution finale trouvée et le temps d'exécution. En effet, bien que BILIN_5_06 fournisse de meilleures solutions que Minamo pour *Rastrigin 10D* (Figure 7(c)) et ne détériore pas la solution pour *G2 plog 10* (Figure 8(c)), le temps d'exécution est toujours plus élevé que les versions BILIN_4_05 et BILIN_4_06 et il est plus élevé que Minamo pour les problèmes *Rosenbrock 10D* (Figure 9(b)) et *G10* (Figure 10(b)). De plus, la version PROP_5_06 est également plus lente que PROP_4_05 et PROP_4_06. De même, elle est plus lente que Minamo pour *Rosenbrock 10D* (Figure 9(b)) et *G10* (Figure 10(b)). Pour finir, elle détériore plus la solution pour *G2 plog 10*, *G10*, *Rastrigin 10D* et *Rosenbrock 10D* bien qu'elle résolvait mieux *G7* que PROP_4_05 (8(a)).

5.4 Résultats pour l'AG intégré dans la boucle *SBO*

Comme nous l'avons dit dans la section précédente, par un souci de temps d'exécution et de non détérioration de la solution finale avec l'AG pur, nous avons exécuté les versions suivantes avec l'AG intégré dans la boucle *SBO* :

- BILIN_4_05 : *GAVaPS* avec l'allocation bi-linéaire pour le calcul de l'espérance de vie, l'espérance de vie maximale EV_{max} fixée à 4 et le taux de reproduction ρ valant 0.5 ;
- BILIN_4_06 : *GAVaPS* avec l'allocation bi-linéaire pour le calcul de l'espérance de vie, EV_{max} fixé à 4 et ρ valant 0.6 ;
- PROP_4_05 : *GAVaPS* avec l'allocation proportionnelle pour le calcul de l'espérance de vie, EV_{max} à 4 et ρ valant 0.5 ;
- PROP_4_06 : *GAVaPS* avec l'allocation proportionnelle pour le calcul de l'espérance de vie, EV_{max} fixé à 4 et ρ valant 0.6.

Chacune de ces versions a été exécutée, par trente *runs* différents, sur les six problèmes d'optimisation. La Figure 5.1 recense la convergence finale des différentes méthodes sur ces problèmes. Les valeurs de convergence pour tous ces problèmes sont semblables à Minamo pour les problèmes d'optimisation de *Rosenbrock* en 2D et en 10D (Figures 5.1(a) et 5.1(b)), le problème *G10* (Figure 5.1(e)) et *G2 plog 10* (Figure 5.1(f)). Cependant, les résultats du problèmes *Rastrigin 10D* (Figure 5.1(c)) sont meilleurs que Minamo, en particulier pour les versions contenant le calcul proportionnel (PROP_4_05 et PROP_4_06). De plus, pour le problème *G7* (Figure 5.1(d)), les versions BILIN_4_06 et PROP_4_06 ont des résultats semblables à Minamo, mais cela n'est pas le cas pour les versions BILIN_4_05 et PROP_4_05. Ainsi, dans la Figure 5.2, le profil fin de performance et le profil fin de données nous démontrent clairement que la version PROP_4_06 est la plus efficace et la plus robuste des versions. De plus, toutes ces versions testées sont meilleures que Minamo.

La Figure 5.3 synthétise le temps d'exécution des différentes méthodes sur les différents problèmes d'optimisation. A l'unanimité, ces versions sont vraiment plus lentes que Minamo. Par exemple, les versions durent dix à trente-cinq fois plus longtemps que Minamo pour le problème *G10* (Figure 5.3(e)). Cependant, la résolution du problème *G7* (Figure 5.3(d)) dure moins longtemps pour les versions BILIN_4_05 et PROP_4_05, sans pour autant avoir des résultats semblables à Minamo (Figure 5.1(d)). De plus, il est clair que la version PROP_4_06, qui est la plus performantes des méthodes pour donner des solutions, est malheureusement la plus lente, pour résoudre tous ces problèmes. Dans ce cas, nous ne pouvons pas retenir une version pour le Chapitre 7, car une telle augmentation du temps d'exécution n'est pas intéressante pour les utilisateurs de Minamo.

Une question peut alors être posée : pourquoi ces versions sont-elles plus lentes que Minamo, en *SBO*, alors que pour l'AG pur cela se sentait sans ressortir aussi clairement ? Nous pouvons affirmer notre intuition faite dans la sous-section précédente, comme maintenant les évaluations de fonctions de l'AG sont peu coûteuses puisqu'elles sont faites sur un méta-modèle. Pour rappel, nous avions dit que la gestion de la méthode de *GAVaPS* est bien trop lourde. Citons par exemple la gestion de la map que nous avons implémentée et qui contient l'espérance de vie de chaque individu.

5.5 Résumé et améliorations suggérées

Nous avons réalisé plusieurs adaptations de cette méthode *GAVaPS* pour l'intégrer dans Minamo. Par exemple, nous avons dû imposer la diminution de la valeur de fitness maximale, car sinon la population courante augmentait de trop et le nombre d'évaluations de fonctions surpassait de loin Minamo. Nous avions également utilisé une map dont les clés étaient des pseudo-identifiants (des valeurs obtenues par hachage des chromosomes) pour stocker l'espérance de vie de chaque individu. Cependant, il fallait gérer les collisions qui arrivaient à cause de l'élitisme de Minamo.

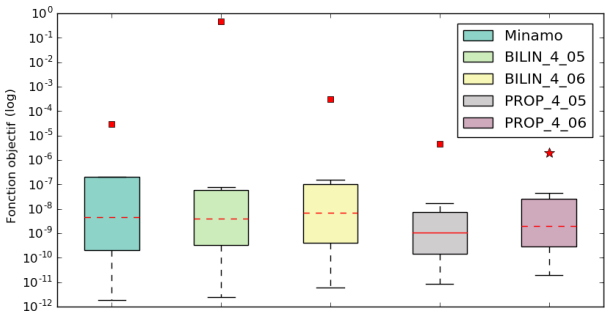
Ensuite, nous avons réalisé une analyse de seize combinaisons de valeurs de deux paramètres, pour l'AG pur : l'espérance de vie maximale EV_{max} et le taux de reproduction ρ. Seize versions de *GAVaPS*, qui utilisaient le calcul bi-linéaire pour l'allocation de l'espérance de vie des individus, ont alors été exécutées sur les six cas tests de base de notre mémoire. Nous avons alors retenu, pour ces versions avec le calcul bi-linéaire, les combinaisons suivantes pour EV_{max} et ρ :

— $EV_{max} = 4$ et $\rho = 0.5$;
— $EV_{max} = 4$ et $\rho = 0.6$;
— $EV_{max} = 5$ et $\rho = 0.6$.

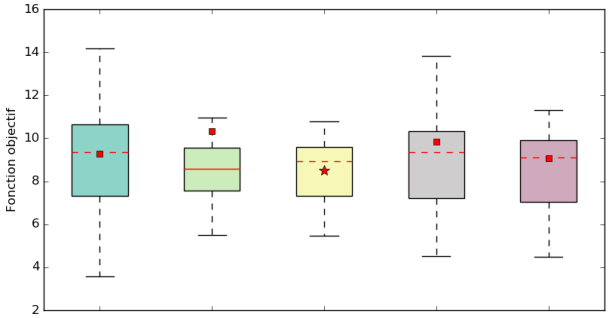
Ces trois combinaisons ont alors été exécutées avec le calcul linéaire et proportionnel pour l'attribution de l'espérance de vie, donnant six versions. Celles-ci ont été comparées avec Minamo et l'allocation bi-linéaire. En conclusion, nous avons retenu pour le Chapitre 7 les combinaisons $(EV_{max}, \rho) = (4, 0.5)$ et $(EV_{max}, \rho) = (4, 0.6)$, avec le calcul proportionnel et bi-linéaire.

Après cela, nous avons alors comparé ces quatre versions avec Minamo, en les exécutant sur l'AG intégré à la boucle *SBO*. Cependant, il y a une nette détérioration du temps d'exécution sur tous les cas tests que nous avons résolus par ces méthodes. Cela est probablement induit par la gestion de cette méthode et des étapes qui la composent, comme par exemple la gestion de la map contenant l'espérance de vie de chaque individu, qui n'apparaissait pas dans la méthode originale.

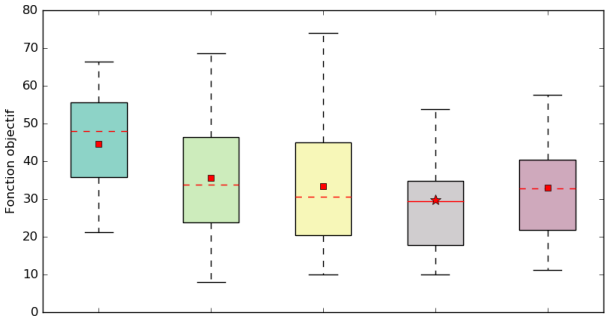
Ainsi, comme améliorations possibles, nous proposons d'enlever cette map, ou d'utiliser des véritables identifiants pour la gérer. Nous pensons également qu'utiliser vingt individus pour créer la population initiales n'est pas suffisant, comme le disent Cook et Tauritz [3], car cela n'induirait pas un assez bon signal au début de l'exécution. La méthode présentée dans le chapitre suivant, *FiScIS-EA*, nécessite plus d'individus au lancement de la recherche.



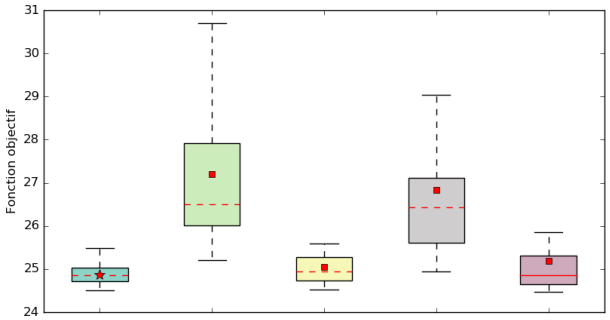
(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif



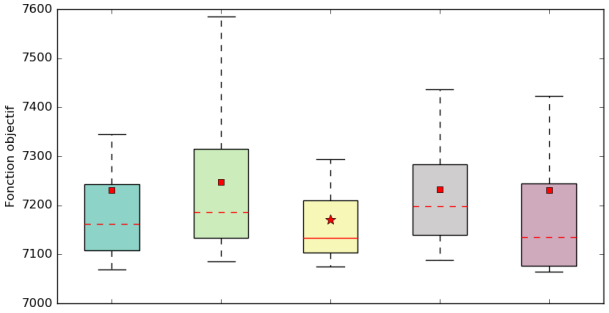
(b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif



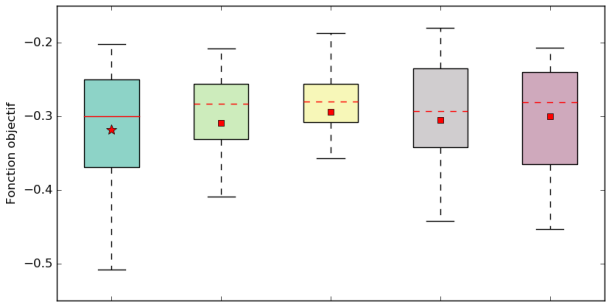
(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif



(d) *G7* - Valeurs finales de la fonction objectif

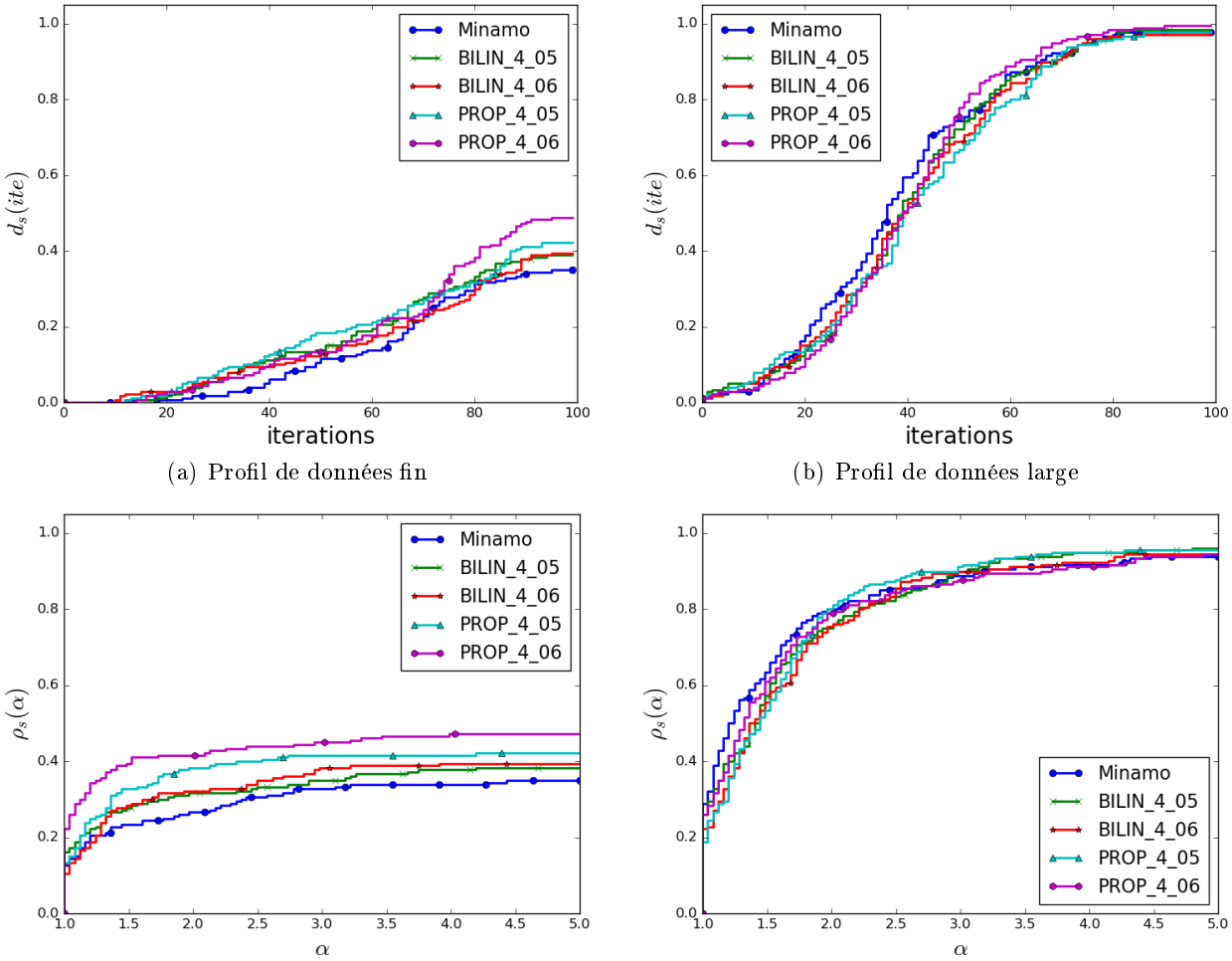


(e) *G10* - Valeurs finales de la fonction objectif



(f) *G2 plog 10* - Valeurs finales de la fonction objectif

FIGURE 5.1 – Ces boîtes à moustache représentent les valeurs finales des quatre versions testées de *GAVaPS* et Minamo sur les cas tests avec l'AG intégré dans une boucle *SBO*. La légende est la même pour toutes ces figures.



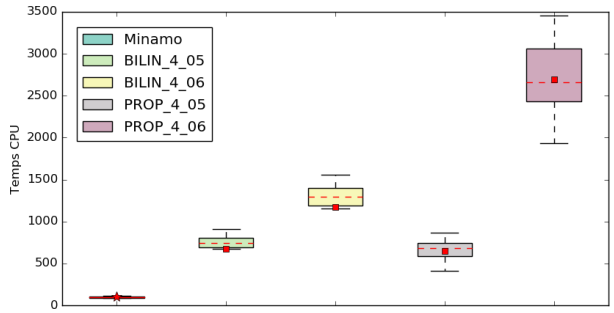
(a) Profil de données fin

(b) Profil de données large

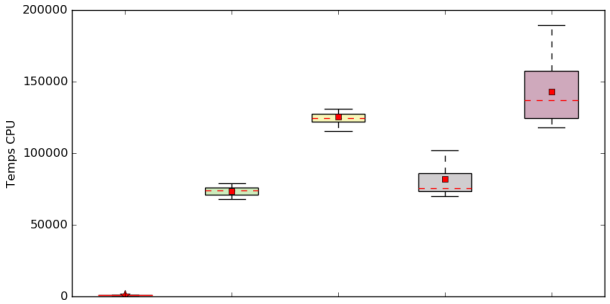
(c) Profil de performance fin

(d) Profil de performance large

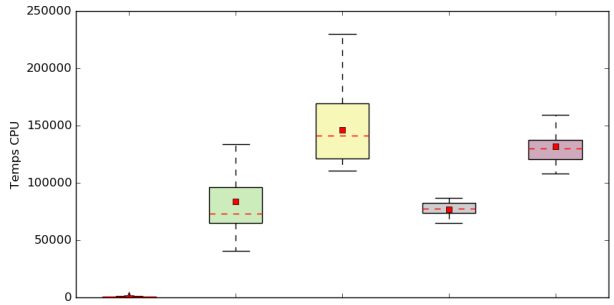
FIGURE 5.2 – Les graphiques de profil de données et de performance sur les six cas tests pour les méthodes *GAVaPS* exécutées au sein de la boucle *SBO*.



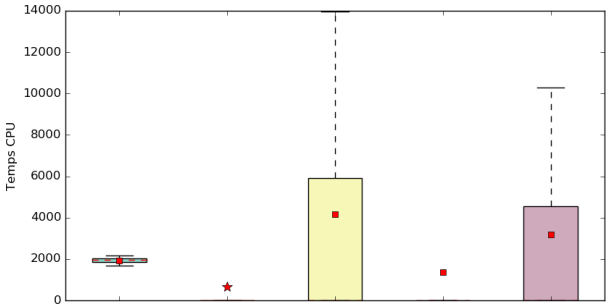
(a) *Rosenbrock 2D* - Temps d'exécution



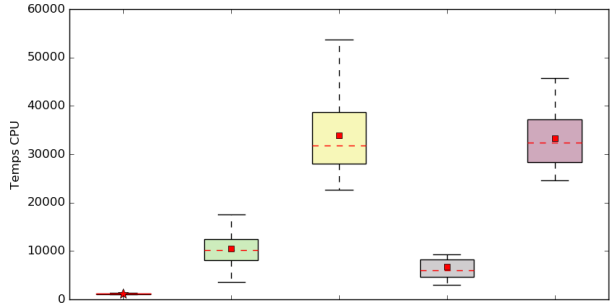
(b) *Rosenbrock 10D* - Temps d'exécution



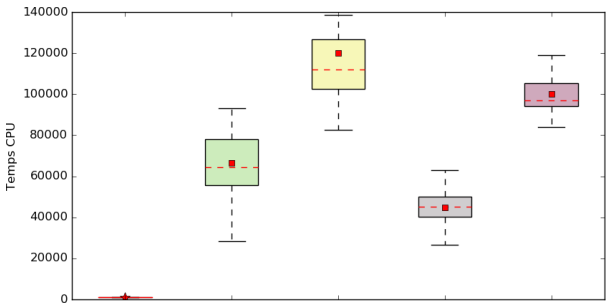
(c) *Rastrigin 10D* - Temps d'exécution



(d) *G7* - Temps d'exécution



(e) *G10* - Temps d'exécution



(f) *G2 plog 10* - Temps d'exécution

FIGURE 5.3 – Ces boîtes à moustache représentent le temps d'exécution en seconde des quatre versions testées de *GAVaPS* et Minamo sur les cas tests avec l'AG intégré dans une boucle *SBO*. La légende est la même pour toutes ces figures.

Chapitre 6

FiScIS-EA - explication, modifications et résultats

Dans ce chapitre, nous allons développer une dernière méthode permettant d'adapter la taille de la population. Celle-ci est nommée *FiScIS-EA*, signifiant en anglais *Fitness Scaled Individual Survival Evolutionary Algorithm*. Elle a été présentée brièvement dans un chapitre antérieur (voir Section 2.1.1) et elle a été introduite dans la litérature par Cook et Tauritz [3]. Nous allons d'abord décrire la méthode originale qui a été développée dans leur article. Pour rappel, celle-ci détermine à chaque itération la probabilité de survie de chaque individu selon un certain calcul. Ensuite, dans une section supplémentaire, nous allons introduire différentes manières de calculer la probabilité de survie en se basant sur les formules de *GAVaPS*, car à premier abord celles-ci sont plus adaptées lorsque beaucoup d'individus sont proches de la valeur minimale de fitness de la population. Puis, comme à notre habitude, nous allons présenter l'implémentation et les adaptations réalisées dans Minamo. Après cela, nous allons comparer les résultats de ces différentes méthodes avec Minamo sur les six cas tests présentés dans la Section 3.4, sur l'AG pur et sur l'AG intégré dans une boucle *SBO*. Une seconde adaptation a été réalisée, en s'inspirant du principe des dents de scie. Celle-ci permet de décroître, non plus linéairement, mais selon la probabilité de survie. Nous expliquerons cette méthodologie. Puis nous la comparerons avec Minamo en utilisant des exécutions de l'AG pur et de l'AG intégré au sein d'une boucle *SBO*, selon les six même cas tests. Pour finir, nous pourrons résumer cette section en retenant les meilleures versions.

6.1 Méthode originale

Commençons par expliquer la méthode originale présentée dans l'article de Cook et Tauritz [3]. En partant du constat que les deux méthodes précédentes rajoutent des paramètres (et qu'il faut les contrôler à leur tour), les auteurs désirent introduire une méthode sans paramètres ajoutés, qui va pouvoir déterminer elle-même la taille de la population, selon l'état de la recherche. De plus, selon ces auteurs, *GAVaPS* qui introduit EV_{max} et EV_{min} fixe implicitement une valeur de convergence pour la taille de la population. Ici, cette méthode supprime des individus à chaque itération. Un individu au sein d'une population possède une chance de survivre pour l'itération suivante. Ainsi, la génération de l'itération suivante contient une partie des individus présents dans la génération actuelle et une partie des enfants générés à l'itération actuelle. Il n'y a pas de sélection d'enfants pour la génération suivante ni même de remplacement de génération.

En réalité, cette chance est une probabilité calculée par la formule suivante pour chaque individu i de $P(t)$ à l'itération t :

$$P_{surv}(i) = \begin{cases} \dfrac{fit(i) - fit_{min}(t)}{fit_{max}(t) - fit_{min}(t)} & \text{si } fit_{min}(t) \neq fit_{max}(t) \\ 1 & sinon, \end{cases} \tag{6.1}$$

où $fit_{min}(t)$ et $fit_{max}(t)$ sont à nouveau les valeurs de fitness minimales et maximales observées dans la population $P(t)$. Ainsi, d'un point de vue algorithmique, pour un nombre aléatoire q tiré entre zéro et un, si celui-ci est plus petit que $P_{surv}(i)$ alors l'individu i survit, sinon il est supprimé de la population.

Cette formule implique instantanément que le meilleur individu de la population est gardé pour la génération suivante ; tandis que le pire individu est supprimé. De plus, de bons individus ont plus de chance de survivre pour la génération suivante. Cependant, certains vont quand même être supprimés. Ceci ne pose pas de problème : cela permet même de supprimer des individus qui sont trop proches, comme il est inutile d'avoir plein d'individus proches, car sinon la recherche est ralentie par manque de variabilité. De plus, par une telle approche, des individus moins bons peuvent également survivre. Ceci implique de pouvoir maintenir une certaine diversité au sein de la population.

De plus, Cook et Tauritz [3] expliquent que cette formule permet de croître le nombre d'individus dans la population quand ceux-ci convergent dans une zone et sont bloqués dans celle-ci. De plus, lorsqu'une descente rapide vers un minimum est en cours, alors l'écart entre le moins bon individu et le meilleur en terme de valeur de fitness est conséquent, puisque fit_{min} décroît rapidement. Ainsi les meilleurs individus survivent plus facilement, s'ils sont proches de fit_{min}, tandis que les moins bons sont aussi supprimés plus facilement. Dès lors, cela permet de réduire la taille de la population et la méthode converge plus aisément au fond du puits.

Comme premier paramètre, nous avons la taille de la population initiale μ_0. Comme celle-ci est obligatoire pour toutes les méthodes, nous ne pouvons pas véritablement la considérer comme un nouveau paramètre introduit par la méthode. Dans l'article de Cook et Tauritz [3], la taille de la population initiale est élevée, car cela permet d'avoir un meilleur signal pour le début de la recherche ; sinon, la population d'individus risque de converger prématurément dans un minimum local ou aura du mal à trouver de bons individus, comme *GAVaPS* nous l'a également démontré dans le chapitre précédent. Comme second paramètre, il y a le nombre d'enfants $\lambda(t)$ à générer à chaque itération. Dans leur article, ils l'ont fixé pour toutes les itérations de l'AG selon les problèmes d'optimisation à différentes valeurs, sans suivre une règle spécifique. Ce nombre λ est élevé (par exemple 250, 300 350). Par contre, dans Minamo, le nombre d'enfants à l'itération t est toujours le nombre d'individus dans la population $P(t)$. Nous ferons de même pour cette méthode. En effet, cela a également l'avantage que si l'algorithme n'a besoin que d'une trentaine d'individus, pourquoi devrait-il créer 250 enfants (par exemple).

Dans la section suivante, nous allons introduire deux nouvelles formules, qui s'inspirent de l'allocation bi-linéaire de *GAVaPS*, et deux interprétations concernant le tirage de nombres aléatoires.

6.2 Calculs différents

Dans l'article de Cook et Tauritz [3], le tirage de nombres aléatoires qui permettrait de les comparer avec la probabilité de survie calculée pour chaque individu n'est pas clairement abordé et détaillé. Aussi bien que nous ne savons pas avec certitude :

- si un nombre aléatoire q uniformément distribué entre zéro et un est tiré pour une génération donnée et si les individus qui ont leur probabilité de survie plus petite que q sont supprimés ;
- ou si ce nombre est tiré aléatoirement pour chaque individu.

Cependant, ils argumentent leur méthodologie en précisant que les individus survivent de manière aléatoire, que de bons individus peuvent mourir tandis que de mauvais peuvent être gardés. Ainsi, nous pensons que la méthode originale de *FiScIS-EA* tire pour chaque individu un nombre aléatoire et le compare à la probabilité de survie calculée. Cependant, nous allons quand même tester les deux stratégies, d'autant plus que le tirage aléatoire pour chaque individu ne sera pas concluant.

La première approche ci-dessus permettrait de réduire fortement la taille de la population si ce nombre tiré est proche de 1 (par exemple 0.9). En effet, cela signifierait que tous les individus qui ont leur probabilité de survie en dessous de 0.9 sont supprimés. Ceci peut être un avantage quand la population croîtra de trop. De plus, bien que la deuxième approche est plus lente puisque, entre autre, il faut tirer un nombre aléatoire pour chaque individu, celle-ci permet de garder de moins bons individus. Ainsi, nous avons implémenté ces deux approches. La première méthode portera le tag **fx** pour fixe et la seconde **vr** pour variable.

Ensuite, nous avons également réalisé une adaptation concernant le calcul de la probabilité de survie pour chaque individu. En effet, à bien regarder, le calcul de la probabilité de survie de *FiScIS-EA* ressemble à la fraction présente dans l'allocation linéaire de *GAVaPS* (Section 5.1). Ainsi, nous avons introduit deux calculs supplémentaires qui s'inspirent de l'allocation bi-linéaire de *GAVaPS*, comme celle-ci nécessitait moins d'individus durant l'exécution, comme nous avons pu le constater dans le chapitre précédent.

1. Le premier calcul utilise également la moyenne. La probabilité de survie pour un individu i est calculée comme suit :

$$p_{surv}(i) = \begin{cases} 0.5 + \frac{1}{2}\left(\frac{fit_{moy} - fit(i)}{fit_{moy} - fit_{min}}\right) & \text{si } fit(i) \leq fit_{moy} \\ \frac{1}{2}\left(\frac{fit_{max} - fit(i)}{fit_{max} - fit_{moy}}\right) & \text{sinon,} \end{cases} \tag{6.2}$$

   où fit_{min}, fit_{max} et fit_{moy} sont la valeur de fitness minimale, maximale observée et la moyenne dans la population respectivement, pour une certaine itération donnée. Notons qu'il faut toujours imposer que la probabilité de survie soit égale à 1 si $fit_{min} = fit_{max}$, sinon il a une division par zéro. Cette approche portera le tag **mo** pour moyenne.

2. Le second calcul utilise la médiane au lieu de la moyenne dans sa formule. La probabilité de survie pour un individu i est alors calculée comme suit :

$$p_{surv}(i) = \begin{cases} 0.5 + \frac{1}{2}\left(\frac{fit_{med} - fit(i)}{fit_{med} - fit_{min}}\right) & \text{si } fit(i) \leq fit_{med} \\ \frac{1}{2}\left(\frac{fit_{max} - fit(i)}{fit_{max} - fit_{med}}\right) & \text{sinon,} \end{cases} \tag{6.3}$$

   où fit_{med} est la valeur de fitness médiane présente dans la population pour une certaine itération donnée. Notons qu'il faut ici imposer que la probabilité de survie soit égale à 1 si $fit_{med} = fit_{min}$ ou 0 si $fit_{med} = fit_{max}$, sinon il a une division par zéro. Cette approche portera le tag **me** pour médiane.

Le calcul original basé sur L'Équation 6.1 portera, quant à lui, le tag **or**.

Dans la section suivante nous allons présenter l'implémentation dans Minamo avec sa légère adaptation concernant une condition à ajouter pour diminuer la valeur de fitness maximale, comme nous l'avons réalisé dans *GAVaPS*, car sinon les calculs sont induits en erreur si cette valeur augmente durant certaines itérations.

6.3 Implémentation et adaptation dans Minamo

Une même observation qui a été réalisée pour *GAVaPS* peut être faite pour la formulation originale et les adaptations (mo et me) de *FiScIS-EA*. Les enfants qui sont ajoutés dans la population doivent diminuer la valeur de fitness maximale fit_{max}. En effet, pour la formule originale, la probabilité de survie d'un individu augmentera à l'itération suivante si fit_{max} augmente et si la valeur de fitness minimale fit_{min} diminue (voire reste inchangée). En d'autres mots, si pour $t > 0$ nous avons $fit_{max}(t) \geq fit_{max}(t-1)$ et $fit_{min}(t) = fit_{min}(t-1)$ (par exemple), alors pour un individu i qui est présent dans $P(t)$ et $P(t-1)$ sa probabilité de survie augmente car

$$\frac{fit(i) - fit_{min}(t)}{fit_{max}(t) - fit_{min}(t)} \geq \frac{fit(i) - fit_{min}(t-1)}{fit_{max}(t-1) - fit_{min}(t-1)}.$$

Ceci ne devrait pas être le cas. De plus, pour les formules contenant une moyenne ou une médiane (mo ou me), le même raisonnement fait pour l'allocation bi-linéaire de *GAVaPS* peut être tenu.

Nous imposerons donc une diminution de la valeur de fitness maximale au cours des itérations. Ceci est réalisé en n'ajoutant que les enfants qui ont une valeur de fitness plus petite que la valeur de fitness maximale de la population courante, comme cela a été implémenté dans le chapitre de GAVaPS (Chapitre 5).

De plus, nous devrons utiliser une population supplémentaire dans Minamo (une population de survie) qui contient les individus gardés d'itérations en itérations. En effet, dans la version originale de Minamo, la population est remplacée par les enfants créés. Il faut donc utiliser la même adaptation déjà décrite dans la Section 5.2.

En résumé, l'AG contenant la stratégie *FiScIS-EA* peut être implémenté par l'Algorithme 4. Notons que nous devons également garantir que la population ne contienne pas moins d'individus que la taille minimale autorisée. En effet, nous nous sommes rendus compte que certaines exécutions s'arrêtaient à cause d'un manque d'individus. Ce qui avait déjà été le cas pour *GAVaPS* et a été détaillé dans la Section 5.2.

6.4 Résultats pour l'AG pur

Comme nous avons pu le comprendre, nous avons trois formules pour calculer la probabilité de survie et deux manières pour gérer le nombre aléatoire tiré q pour déterminer quels individus survivent et quels sont ceux qui décèdent. Nous avons testé les six combinaisons possibles comme des versions différentes. Le Tableau 6.1 reprend les tags qui correspondent aux différentes combinaisons de paramètres et seront présents dans les graphiques de cette section et la suivante.

La convergence finale selon les différentes versions est reprise dans la Figure 6.1 avec des boîtes à moustache. De manière globale, en comparaison avec Minamo, certaines méthodes implémentées améliorent la convergence finale pour certains cas tests et d'autres la détériorent pour d'autres cas tests. Les versions de *FiScIS-EA* où le nombre aléatoire tiré q est fixe pour une génération donnée (versions **fx** : fx_or, fx_mo et fx_me) permettent bien souvent une amélioration. L'amélioration est très avantageuse pour l'unique problème uni-modale (*Rosenbrock 2D*). De plus, pour les problèmes avec contraintes (*G2 plog 10*, *G7* et *G10*), une amélioration est également observable. Cependant, pour le problème *Rosenbrock 10D*, une détérioration de la solution est à déplorer et, pour le test *Rastrigin 10D*, les solutions finales sont identiques (selon la médiane) à Minamo mais celles-ci sont moins variables. Ceci est un avantage. D'un autre côté, les versions où le nombre aléatoire est tiré pour chaque individu (versions **vr** : vr_or, vr_mo et vr_me) n'améliorent guère les solutions finales. En effet, nous constatons uniquement une amélioration pour les problèmes sans contraintes

Algorithme 4 : *FiScIS-EA* implémenté dans Minamo

début
  $t = 0$;
  initialiser $P(t)$ (qui sera considéré comme les parents) ;
  évaluer $P(t)$;
  tant que $t < 100$ **faire**
    si $t > 0$ **alors**
      pour tous les $i_E \in E(t-1)$ *tq* $fit(i_E) \geq fit_{max}(t-1)$ **faire**
        ajouter i_E dans l'ensemble I_{sup} ;
      supprimer de $E(t-1)$ les individus ayant les indices dans I_{sup} ;
      vider l'ensemble I_{sup} ;
      $P'(t) \leftarrow P'(t-1) \cup E(t-1)$;
      tirer q si version **fx** ;
      calcul des données utiles selon la version (fit_{min}, fit_{moy}, fit_{med} et/ou fit_{med} selon **or**, **mo** ou **me**) ;
      pour tous les *individus i de* $P'(t)$ **faire**
        calculer $P_{surv}(i)$ suivant une des trois formules (**or**, **mo** ou **me**) ;
        tirer q si version **vr** ;
        si $q \geq P_{surv}(i)$ **alors**
          ajouter i dans l'ensemble I_{sup} ;
      si $\mu'(t-1) - \mid I_{sup} \mid > \mu_{min}$ **alors**
        supprimer de $P'(t)$ les individus ayant les indices dans I_{sup} ;
      vider l'ensemble I_{sup} ;
      $P(t) \leftarrow P'(t)$;
    sinon
      $P'(0) \leftarrow P(0)$
    croiser les individus de $P(t)$ pour obtenir les enfants $E(t)$ (au même nombre) ;
    muter les enfants $E(t)$;
    évaluer les enfants $E(t)$;
    $t \leftarrow t + 1$;

Tag	Tirage	Formule
fx_or	fixe	originale (Équation 6.1)
fx_mo	fixe	moyenne (Équation 6.2)
fx_me	fixe	médiane (Équation 6.3)
vr_or	variable	originale (Équation 6.1)
vr_mo	variable	moyenne (Équation 6.2)
vr_me	variable	médiane (Équation 6.3)

TABLE 6.1 – Ce tableau reprend les tags des différentes combinaisons de paramètres pour *FiScIS-EA*.

de haute dimension, à savoir pour *Rastrigin 10D* et *Rosenbrock 10D* (sauf la version avec le calcul original pour ce dernier). Pire encore, nous observons une détérioration de la solution finale pour *G7*, *G10* et même *Rosenbrock 2D* par rapport à Minamo, tandis que pour *G2 plog 10*, la solution finale semble identique à Minamo.

Les temps d'exécution des différentes méthodes implémentées sont présentés dans la Figure 6.2. Globalement, un gain du temps d'exécution par rapport à Minamo peut être observé pour les

différents cas tests, avec certaines méthodes plus rapides que d'autres. Les méthodes où le nombre aléatoire q est tiré une unique fois pour tous les individus d'une génération donnée (fx) améliore bien souvent le temps d'exécution de l'AG en comparaison avec Minamo. Pour *G7*, le temps d'exécution est réduit de moitié. Pour *Rosenbrock 2D* et *G2 plog 10*, le temps d'exécution est divisé par deux voire par trois. L'amélioration la plus notable est pour *Rastrigin 10D* où la durée d'exécution est divisée par quatre. Notons que pour *G10* et *Rosenbrock 10D* le temps d'exécution reste semblable à Minamo. A côté de cela, les versions *FiScIS-EA* qui tirent un nombre aléatoire pour chaque individu (vr) n'améliorent pas autant le temps d'exécution, sauf pour le problème *G2 plog 10* où la version avec le calcul médian pour l'espérance de vie diminue encore plus la durée d'exécution. Remarquons également que la version originale pour le calcul de la probabilité de survie est considérablement plus lente que Minamo et les autres versions de *FiScIS-EA*. Ceci est observable pour les problèmes avec contraintes *G7* et *G10*.

De plus, concernant les graphes d'évolutions moyens qui sont repris dans la Figure 11 de l'Annexe B, plusieurs phénomènes spécifiques pour trois cas tests différents sont à noter, tandis que les autres cas tests ont des évolutions semblables entre les différentes versions testées. Premièrement, pour *Rosenbrock 2D*, les méthodes avec le nombre aléatoire tiré pour chaque individu (vr) font plonger la convergence au cours des itérations vers l'unique minimum (qui est en zéro). Cela se remarque pour la version du calcul de la probabilité de survie avec la formulation contenant la valeur médiane pour les fitness (vr_me), suivi de la formule contenant la moyenne (vr_mo) et en dernier la formule originale (vr_or). Deuxièmement, pour *Rastrigin 10D*, toutes les méthodes implémentées de *FiScIS-EA* convergent rapidement : après cinquante itérations la recherche n'avance guère. Par contre, Minamo nécessite plus d'itérations pour commencer à décroître la valeur de la fonction objectif sans véritablement atteindre des solutions semblables aux versions implémentées. Pour finir, il en est de même pour *G2 plog 10*.

Pour finir, dans la Figure 6.3, nous avons repris le nombre d'évaluations de fonctions de toutes les exécutions, sous forme de boîtes à moustache, pour chaque problème d'optimisation. Premièrement, les mêmes comparaisons faites entre les méthodes pour le temps d'exécution peuvent être réalisées pour le nombre d'évaluations. En effet, nous constatons bien que le nombre d'évaluations de fonctions influence le temps d'exécution. Par exemple, pour *Rosenbrock 2D*, nous avons que :

- les versions où le nombre aléatoire q est fixe (fx_me, fx_mo et fx_or) nécessitent moins d'évaluations de fonctions par rapport aux versions où ce nombre est tiré pour chaque individu (vr_me, vr_mo et vr_or), comme le montre la Figure 6.3(a) ;
- le temps d'exécution est plus lent pour les versions vr_me, vr_mo et vr_or par rapport aux versions fx_me, fx_mo et fx_or, comme le montre la Figure 6.3(a).

Ainsi, il y a bien une corrélation entre le nombre d'évaluations de fonctions et la durée de l'exécution. Ainsi, cela peut également expliquer la lenteur des méthodes où le nombre q est tiré pour chaque individu (vr_me, vr_mo et vr_or). De même, la version contenant le calcul original pour la probabilité de survie de chaque individu (vr_or) est la méthode qui utilise le plus d'individus. C'est pour cela que son temps d'exécution n'était pas assez amélioré par rapport à Minamo.

Le Tableau 6.2 synthétise les différentes observations qui ont été faites. Nous nous focalisons sur la convergence finale et la durée d'exécution pour voir s'il y a une différence entre Minamo et les six versions implémentées. Une gradation entière de "2" à "-2" part d'une nette amélioration à une nette détérioration par rapport à Minamo. Certes cette façon de faire est subjective, nous devrions faire des tests statistiques plus poussées pour vérifier que, statistiquement, les résultats sont bien différents. Une somme des poids pour chaque méthode a été réalisée. Celle-ci nous permet de mettre en avant la version fx_me et fx_mo pour le chapitre comparatif des différentes stratégies. De plus, la version originale du calcul de la probabilité de survie allié à un tirage aléatoire du nombre q pour chaque individu (vr_or) est la version la moins bonne sur les six. Cela nous prouve qu'il faut tirer

un unique nombre aléatoire pour tous les individus d'une génération donnée et cela démontre que la formule originale de l'article n'est peut-être pas la plus adaptée.

		fx_me	fx_mo	fx_or	vr_me	vr_mo	vr_or
Rosenbrock 2D	(conv.)	2	2	1	-2	-2	-1
	(durée)	2	2	2	1	1	1
Rosenbrock 10D	(conv.)	-1	-1	-1	1	1	-1
	(durée)	0	0	0	0	-2	-1
Rastrigin 10D	(conv.)	0	0	0	1	1	1
	(durée)	2	2	2	2	1	1
G7	(conv.)	1	1	1	-1	-1	-1
	(durée)	2	2	2	2	2	-2
G10	(conv.)	1	1	0	-1	-1	-2
	(durée)	1	1	-1	1	-1	-2
G2 plog 10	(conv.)	2	2	2	-1	0	0
	(durée)	1	1	1	2	2	1
somme des poids	(conv.)	**5**	**5**	3	-3	-2	-4
	(durée)	**8**	**8**	6	8	3	-2

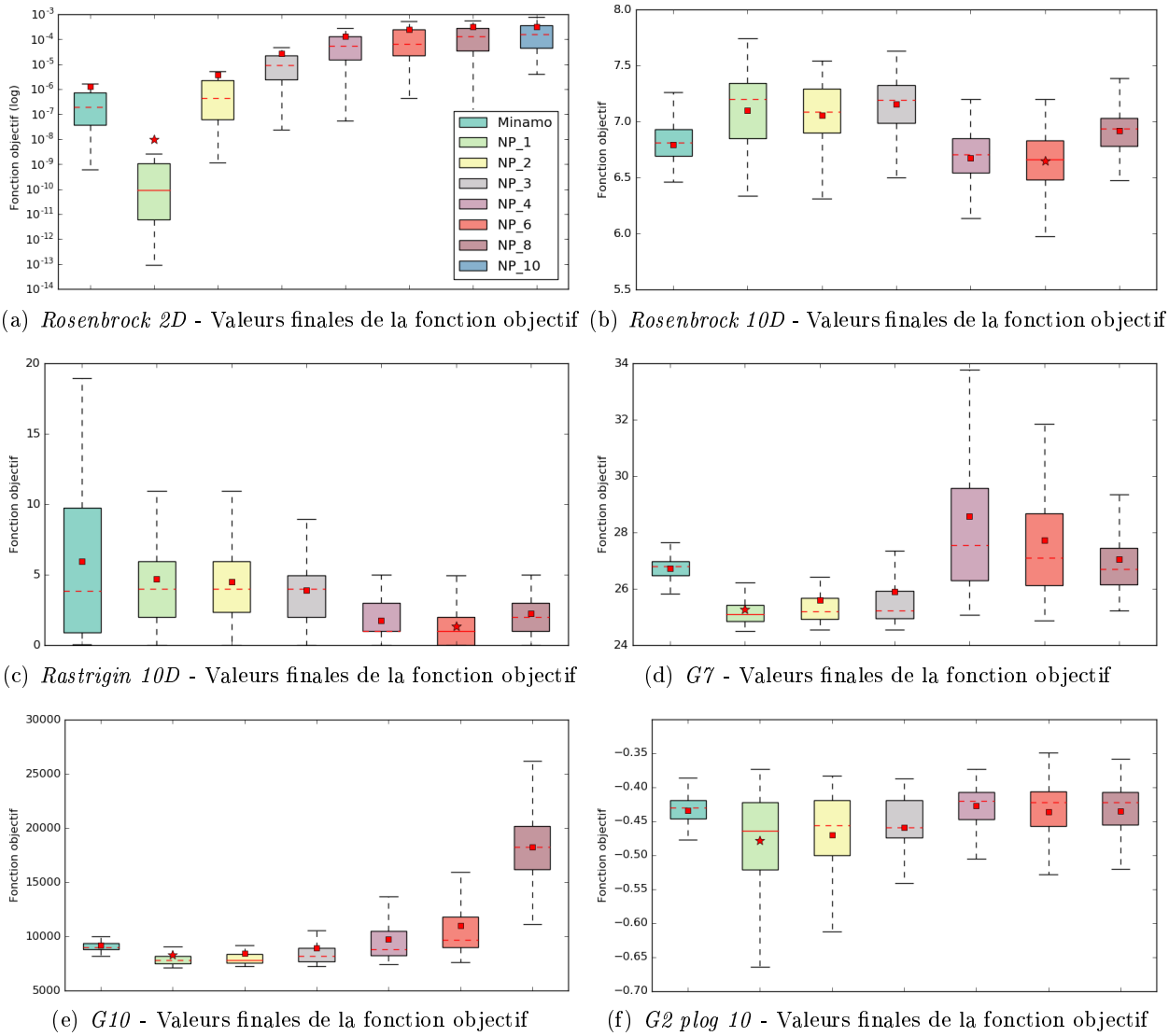
TABLE 6.2 – Ce tableau synthétise les différentes observations sur les résultats des versions de *FiScIS-EA* exécutées sur l'AG pur. L'abréviation "conv." signifie "convergence".

6.5 Résultats pour l'AG intégré dans la la boucle *SBO*

Ces six méthodes ont également été testées sur l'AG inclus dans la boucle *SBO*. Nous allons analyser les résultats des six mêmes cas tests dans cette section.

La Figure 6.4 reprend les solutions heuristiques finales de six cas tests sur les différentes méthodes de *FiScIS-EA* implémentées dans la boucle *SBO*. Nous constatons qu'il n'y a pas d'amélioration franche par rapport à Minamo. En effet, pour certains cas tests les résultats sont véritablement comparables (pour *G2 plog 10* et *Rosenbrock 10D*), tandis que les résultats sont légèrement détériorés pour n'importe quelle version exécutée sur *Rastrigin 10D*. Ceci est également observable dans la Figure 12 de l'Annexe B qui montre l'évolution de la fonction objectif moyenne de chaque version. Cependant, de légères améliorations dans la solution finale sont observées pour *G7* et *G10*. Cela ne peut être conclu que pour les versions de *FiScIS-EA* où le nombre aléatoire, tiré pour être comparé avec la probabilité de survie, est déterminé une unique fois pour tous les individus d'une même génération (fx).

Ensuite, la Figure 6.5 présente le temps d'exécution, des cent exécutions des différentes méthodes sur les différents problèmes d'optimisation. D'une vue globale, nous observons que la durée d'exécution augmente selon la version exécutée : les moins coûteuses en temps de calcul sont les versions où le nombre aléatoire est tiré une unique fois pour tous les individus d'une même génération (versions fx) et les plus coûteuses à l'inverse sont les versions vr. De plus, en ordonnant les calculs des moins coûteux aux plus coûteux nous avons la version avec la médiane (fx_me), suivie de la version avec la moyenne (fx_mo) suivie du calcul original (fx_or). Sinon, plus précisément, les temps de calcul sont augmentés pour toutes les versions sur le problème *Rastrigin 10D* et sur le cas test *G2 plog 10*. Cependant, le temps de calcul est réduit pour les variantes fx sur *Rosenbrock 2D* et *G7*, tandis qu'il reste constant pour *Rosenbrock 10D* et *G10*. De plus, pour les variantes vr, la durée d'exécution est augmentée pour *Rosenbrock* en 2D et en 10D, tandis que les calculs que nous avons introduits (moyenne et médiane) permettent une amélioration du temps de calcul pour *G7* et *G10*.



(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif (b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

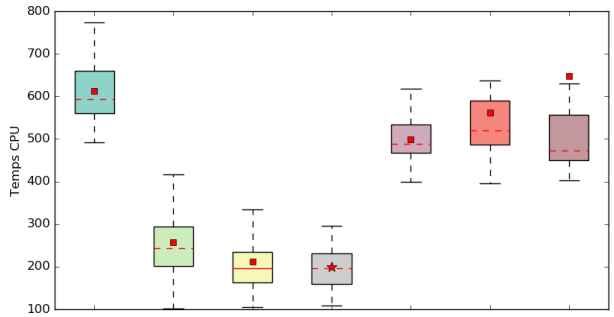
(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif (d) *G7* - Valeurs finales de la fonction objectif

(e) *G10* - Valeurs finales de la fonction objectif (f) *G2 plog 10* - Valeurs finales de la fonction objectif

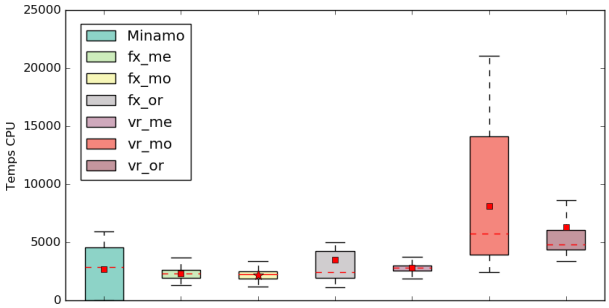
FIGURE 6.1 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* et Minamo sur les six cas tests avec l'AG pur. Celles-ci concernent les valeurs finales des fonctions objectifs. La légende est la même pour toutes ces figures.

Dans la Figure 6.6, les graphes de données et de performance sont répertoriés. A première vue, Minamo est repris parmi toutes les méthodes implémentées, sans se démarquer, pour un seuil fin, tandis qu'il est plus performant dans la recherche superficielle de la solution (seuil large). Sinon, les versions où le nombre q est tiré pour tous les individus d'une génération (fx) avec les calculs nouvellement introduits (me et mo) résolvent plus de problèmes d'itérations en itérations que Minamo pour le seuil fin (Figure 6.6(a)). Ceci confirme notre analyse précédente. De plus, ces deux méthodes sont légèrement plus efficaces que Minamo et plus robuste que Minamo (Figures 6.6(a) et 6.6(c)).

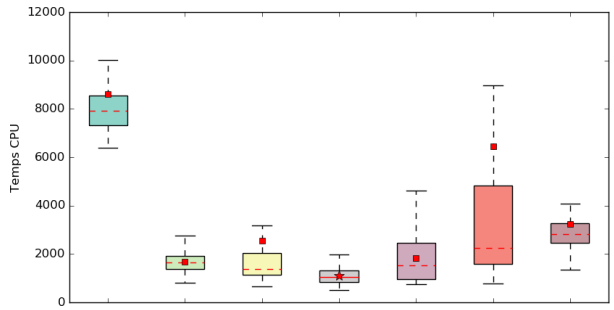
Ainsi, nous retenons *FiScIS-EA* avec le nombre aléatoire tiré une unique fois par génération, associé à la formule médiane ou à la moyenne (versions fx_me et fx_mo), pour le chapitre comparatif (Chapitre 7). Précisons tout de même que le temps de calcul s'alourdit pour les versions vr car le nombre d'individus créés au total dans chaque exécution de l'AG est plus élevé que pour Minamo.



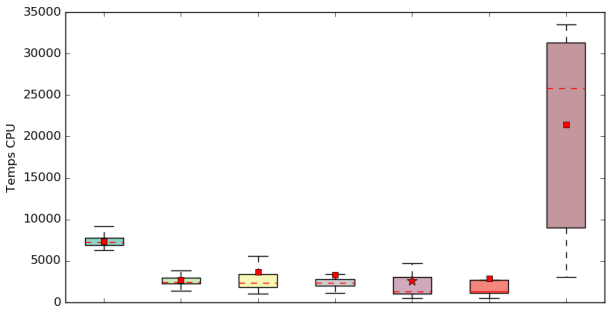
(a) *Rosenbrock 2D* - Temps d'exécution



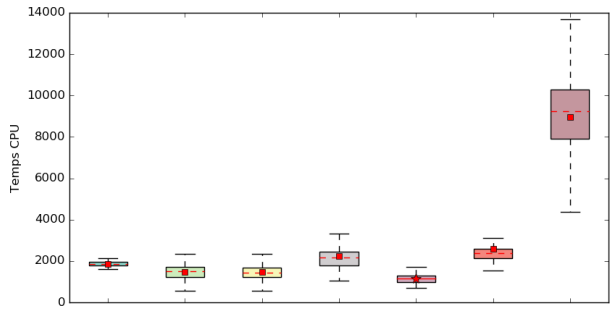
(b) *Rosenbrock 10D* - Temps d'exécution



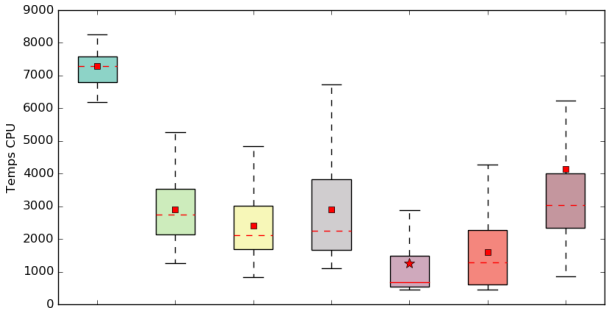
(c) *Rastrigin 10D* - Temps d'exécution



(d) *G7* - Temps d'exécution

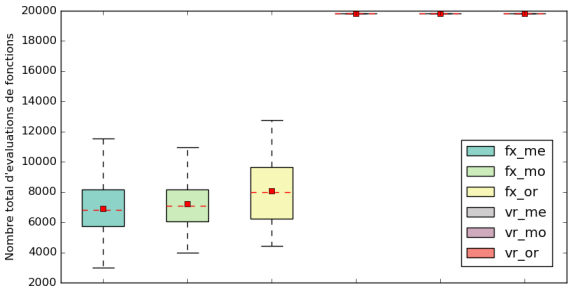


(e) *G10* - Temps d'exécution

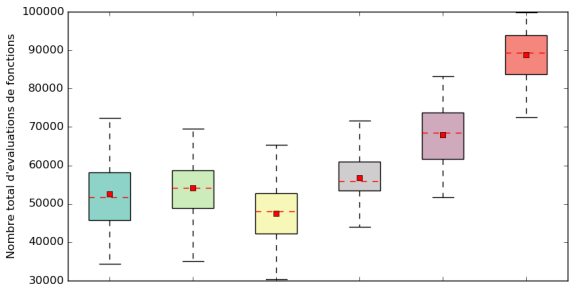


(f) *G2 plog 10* - Temps d'exécution

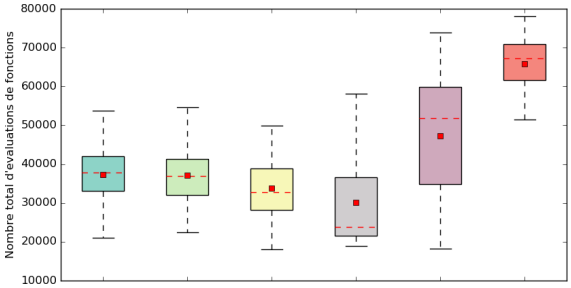
FIGURE 6.2 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* et Minamo sur les six cas tests avec l'AG pur. Celles-ci concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.



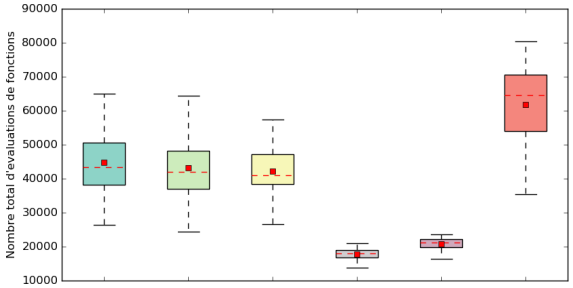
(a) *Rosenbrock 2D* - Nombre d'évaluations de fonctions (Minamo : 20'000 évaluations)



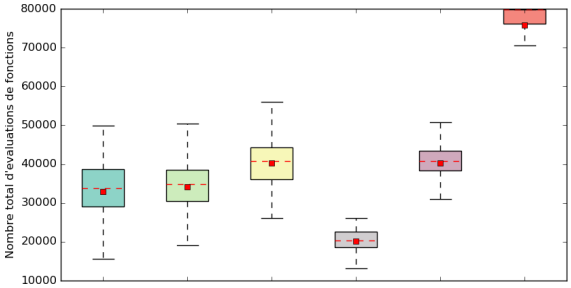
(b) *Rosenbrock 10D* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)



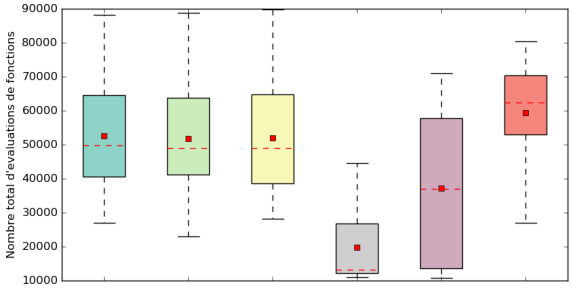
(c) *Rastrigin 10D* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)



(d) *G7* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

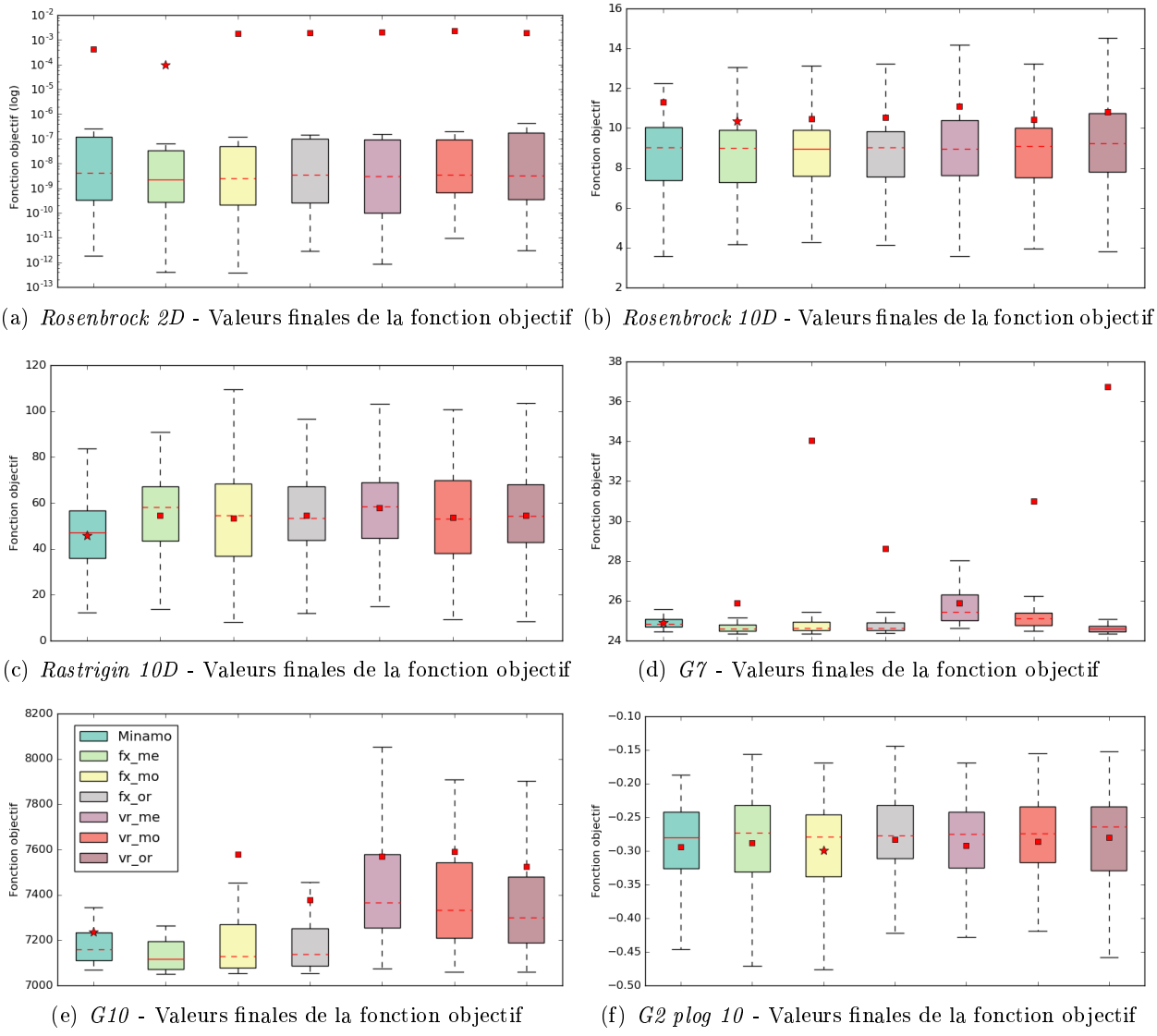


(e) *G10* - Nombre d'évaluations de fonctions (Minamo : 80'000 évaluations)



(f) *G2 plog 10* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

FIGURE 6.3 – Ces graphiques représentent le nombre d'évaluations de fonctions des différentes stratégies *FiScIS-EA* qui ont été exécutées avec l'AG pur sur les différents problèmes. Le nombre d'évaluations de fonctions réalisées par Minamo pour chaque méthode est repris dans la description de chaque problème. La légende est la même pour toutes ces figures.

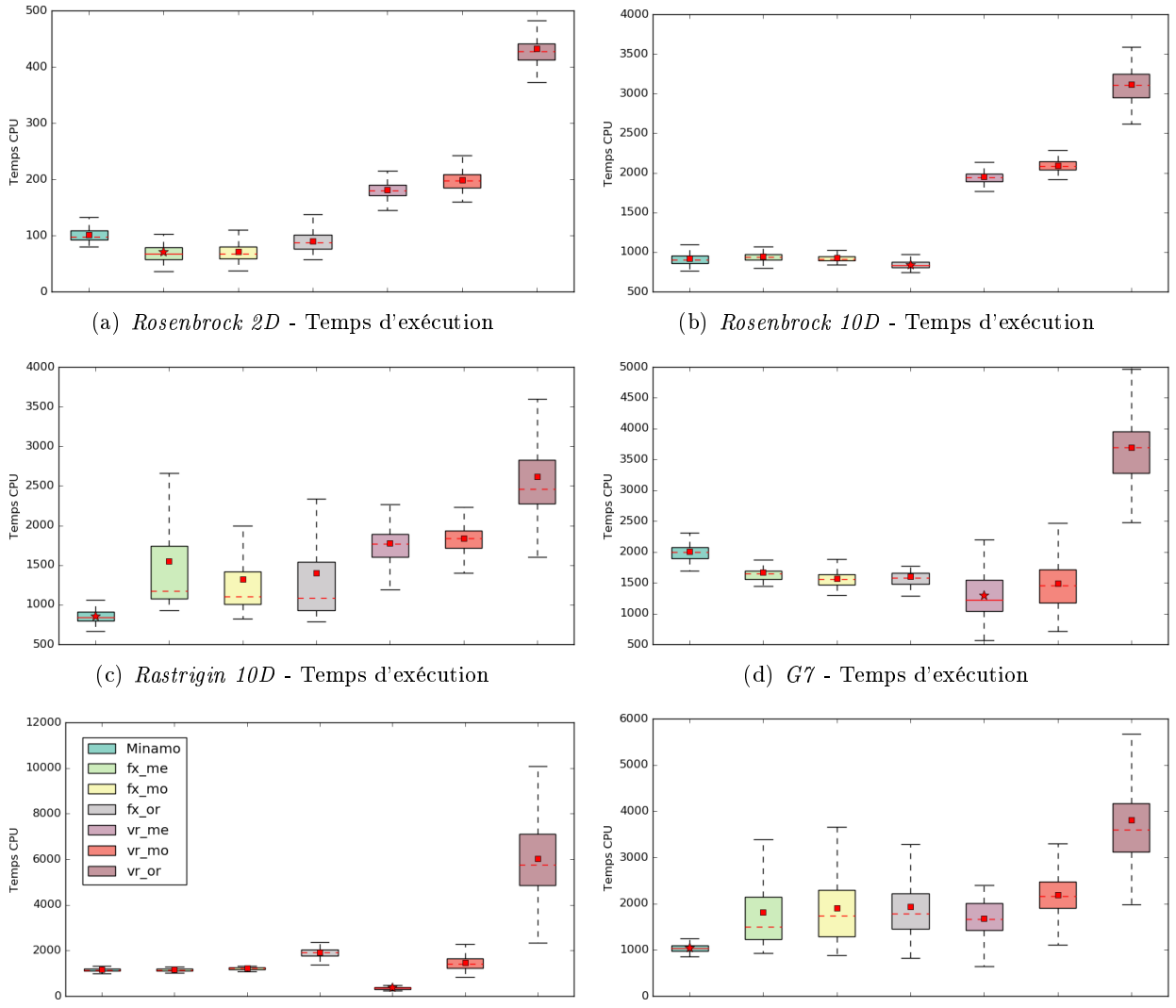


(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif (b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif (d) *G7* - Valeurs finales de la fonction objectif

(e) *G10* - Valeurs finales de la fonction objectif (f) *G2 plog 10* - Valeurs finales de la fonction objectif

FIGURE 6.4 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent les valeurs finales des fonctions objectifs. La légende est la même pour toutes ces figures.



(a) *Rosenbrock 2D* - Temps d'exécution

(b) *Rosenbrock 10D* - Temps d'exécution

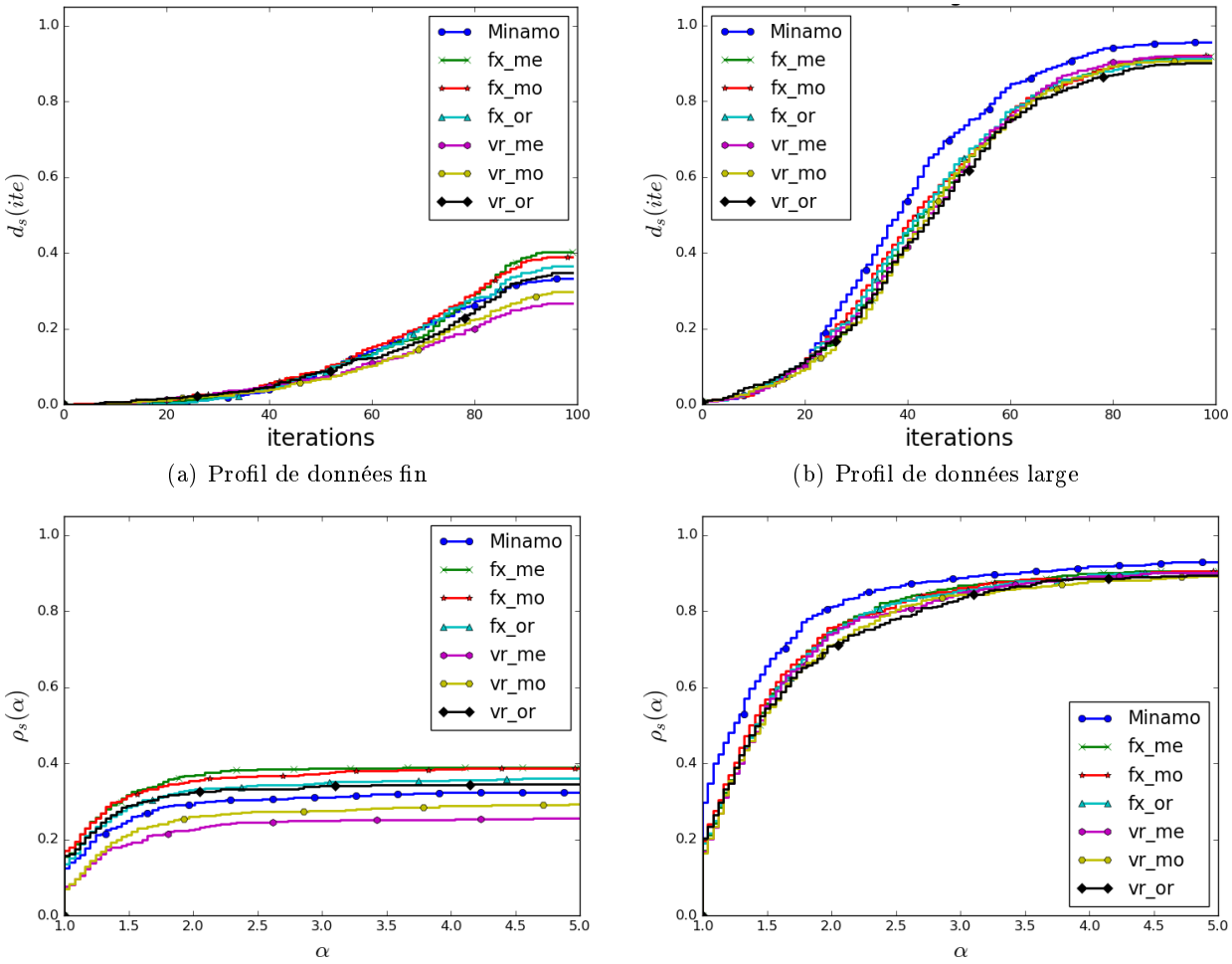
(c) *Rastrigin 10D* - Temps d'exécution

(d) *G7* - Temps d'exécution

(e) *G10* - Temps d'exécution

(f) *G2 plog 10* - Temps d'exécution

FIGURE 6.5 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.



(a) Profil de données fin

(b) Profil de données large

(c) Profil de performance fin

(d) Profil de performance large

FIGURE 6.6 – Les graphiques de profil de données et de performance sur les six cas tests pour les méthodes *FiScIS-EA* exécutées au sein de la boucle *SBO*.

6.6 Adaptation en dent de scie

De la même manière que nous avons appliqué une adaptation en s'inspirant de *GAVaPS* dans les sections précédentes, nous nous sommes également inspirés de *SawTooth* pour réaliser une autre adaptation. En effet, nous avons vu dans le Chapitre 4 qu'il était intéressant de décroître la taille de la population et d'ajouter des individus aléatoirement à des périodes régulières afin de revenir à la taille originale de la population. De plus, nous espérons ainsi diminuer le temps de calcul, car ceci nous assurera la diminution du nombre d'individus créés au total durant les itérations par rapport à Minamo.

Pour réaliser ceci, nous somme repartis du code existant de Minamo et, au lieu de créer une population secondaire pour garder les individus de génération en génération, nous avons gardé la méthode qui remplaçait la population courante par les enfants. Ensuite, après évaluation de ces enfants, ceux-ci sont réduits en utilisant le principe de *FiScIS-EA* avec les méthodes de calcul détaillées en Section 6.2. En d'autres mots, à l'itération t, la population $P(t)$ qui contient $\mu(t)$ individus est croisée pour fournir $\mu(t)$ enfants $E(t)$. Sur ces enfants, nous supprimons ceux qui par leur probabilité de survie doivent décéder. Cependant, quand la suppression de ces individus va décroître excessivement la taille de la population (c'est-à-dire quand la taille de la population va passer en dessous de μ_{min}), alors nous rajoutons des individus dans la population selon la stratégie *LHS* afin de revenir à la taille originale. Cette méthodologie n'impose donc pas à l'avance le nombre de décroissances nécessaire pour les cent itérations.

Notons par contre que nous n'imposons plus la diminution du fit_max. En effet, cela n'aurait plus de sens, dans la mesure où nous ne gardons pas une population dans laquelle des individus survivent. De plus, si nous avions fait cela, alors la population allait décroître rapidement de manière excessive.

Nous notons alors cette variante *FiScIS-EA* en dent de scie (DS). L'implémentation de celle-ci est décrite dans l'Algorithme 5.

6.7 Résultats de l'adaptation en dent de scie

Dans cette section, nous avons donc testé les six versions déployées précédemment mais adaptées avec un comportement en dent de scie comme décrit dans la section précédente. Dans le Tableau 6.3, nous avons repris les différents tags utilisés dans les différents graphiques que nous présenterons ici.

Tag	Tirage	Formule
DS_fx_or	fixe	originale (Équation 6.1)
DS_fx_mo	fixe	moyenne (Équation 6.2)
DS_fx_me	fixe	médiane (Équation 6.3)
DS_vr_or	variable	originale (Équation 6.1)
DS_vr_mo	variable	moyenne (Équation 6.2)
DS_vr_me	variable	médiane (Équation 6.3)

TABLE 6.3 – Ce tableau reprend les tags des différentes combinaisons de paramètres, pour *FiScIS-EA* en dent de scie.

6.7.1 Pour l'AG pur

Selon la Figure 6.7, ces méthodes nouvellement introduites détériorent les résultats finaux par rapport à Minamo. Un phénomène peut être observé pour la majorité des cas tests (*Rosenbrock* en 2D et 10D, *G7* et *G10*) : entre les versions testées de *FiScIS-EA* en dent de scie, les versions avec le

Algorithme 5 : *FiScIS-EA* en dent de scie implémenté dans Minamo

```
1  début
2      t = 0 ;
3      initialiser P(t) ;
4      évaluer P(t) ;
5      tant que t < 100 faire
6          croiser les individus de P(t) pour obtenir les enfants E(t) (au même nombre) ;
7          muter les enfants E(t) ;
8          évaluer les enfants E(t) ;
9          t ← t + 1 ;
10         P(t) ← E(t − 1) ;
11         tirer q si version fx ;
12         calcul des données utiles selon la version (fit_min, fit_moy, fit_med et/ou fit_med selon
             or, mo ou me) ;
13         pour tous les individus i de P(t) faire
14             calculer P_surv(i) suivant une des trois formules (or, mo ou me) ;
15             tirer q si version vr ;
16             si q ≥ P_surv(i) alors
17                 ajouter i dans l'ensemble I_sup ;
18         si μ(t)− | I_sup | > μ_min alors
19             supprimer de P(t) les individus ayant les indices dans I_sup ;
20         sinon
21             ajouter μ_max − μ(t) individus par LHS ;
22         supprimer l'ensemble I_sup ;
```

calcul original de la probabilité de survie sont meilleures que les versions avec le calcul moyen, qui, elles, sont meilleures que les variantes avec la médiane (à l'inverse donc des résultats présentés dans la Section 6.4). De plus, pour les problèmes d'optimisation *G2 plog 10* et *Rastrigin 10D*, toutes les versions nouvellement implémentées sont identiques, avec *Rastrigin 10D* qui possède des résultats plus mauvais pour les versions vr (où un nombre aléatoire q est donc tiré pour chaque individu). Toutes ces remarques sont également confirmées par la Figure 13 de l'Annexe C. Notons tout de même, qu'en moyenne les versions utilisant la formule originale pour le calcul de la probabilité de survie sont équivalentes à Minamo pour *G10*.

Selon la Figure 6.8, le temps d'exécution des différentes méthodes est globalement diminué pour tous les cas tests sauf pour *G10*. Notons également que des versions implémentées sont plus rapides entre elles, notamment les versions où le nombre aléatoire est tiré pour chaque individu.

Ainsi, pour le chapitre comparatif des différentes méthodes retenues (Chapitre 7), nous garderons la version de *FiScIS-EA* en dent de scie qui calcule la probabilité de survie selon la formule originale et qui tire un nombre aléatoire pour chaque individu (DS_vr_or). En effet, celle-ci permet d'obtenir les meilleurs résultats (parmi les versions testées de *FiScIS-EA* en dent de scie) et avec une rapidité accrue.

6.7.2 Pour l'AG intégré dans la boucle *SBO*

Selon la Figure 6.9(c), la convergence est améliorée par rapport à Minamo pour toutes les implémentations de *FiScIS-EA* en dent de scie, pour le problème *Rastrigin 10D*. Sinon, les versions où le calcul de la probabilité de survie est basée sur la formule originale, améliorent ou du moins ne

détériorent pas la solution trouvée par rapport à Minamo, tandis que les autres calculs (moyenne ou médiane) détériorent considérablement les solutions trouvées par rapport à Minamo. Notons que les versions avec le nombre aléatoire tiré pour chaque individu (vr) ou tiré une unique fois pour tous les individus d'une population (fx) ont des solutions semblables pour tous les cas tests, sauf pour *G7* où l'implémentation fx avec le calcul original est la meilleure version de *FiScIS-EA* en dent de scie, tandis que pour *Rosenbrock 10D* c'est l'inverse (l'implémentation vr avec le calcul original est la meilleure). Des observations semblables peuvent être faites pour les graphes moyens d'évolutions des fonctions objectifs présentés dans la Figure 14 de l'Annexe C. Ainsi, d'un point de vue convergence, nous serons tentés de garder les versions DS_fx_or et DS_vr_or.

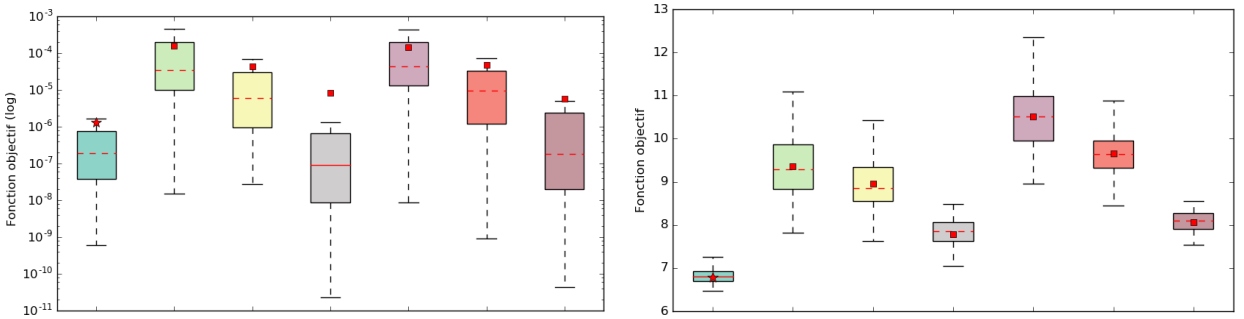
Selon la Figure 6.10, tous les temps d'exécution des méthodes *FiScIS-EA* en dent de scie sont considérablement réduits par rapport à Minamo. Les versions avec le nombre aléatoire tiré q pour chaque individu (vr) sont plus rapides que les versions où q est tiré une unique fois pour une génération donnée (fx). Cela est causé par le fait qu'il est nécessaire de réaliser plus souvent le *LHS* pour combler la population quand q est tiré une unique fois par génération. En effet, cette philosophie supprime plus d'individus comme nous l'avons expliqué dans la Section 6.2. De plus, les versions où le calcul de la probabilité de survie utilise la moyenne (mo) sont plus rapides que les autres. Ainsi, les versions DS_fx_or et DS_vr_or peuvent être gardées aussi pour améliorer le temps d'exécution.

Néanmoins, pour mieux choisir la méthode *SBO* de ces six versions implémentées en dent de scie, nous avons présenté les profils de données et de performance (les graphes de Dolan et More) dans la Figure 6.11. En terme de problèmes résolus, DS_fx_or et DS_vr_or en résolvent plus, en effet, que Minamo au cours des itérations (Figure 6.11(a)) avec un seuil fin, tandis que Minamo reste plus dans la globalité (Figure 6.11(b)). De plus, DS_fx_or et DS_vr_or sont aussi plus robustes que Minamo pour un seuil fin (Figure 6.11(c)).

6.8 En résumé

Concernant les six premières versions testées de *FiScIS-EA*, le temps d'exécution est souvent allongé si le nombre aléatoire q, pour comparer aux espérance de vie des individus, est tiré pour chaque individu. Ceci est dû au fait que cette méthode ne réduit pas assez la valeur de fitness maximale et n'est pas assez catégorique pour exclure les mauvais individus, ainsi trop d'individus sont créés par rapport à Minamo (ou autres méthodes). Pour les exécutions avec l'AG pur ou l'AG intégré dans une boucle *SBO*, les versions qui sont à retenir pour le chapitre suivant, où nous comparerons les différentes méthodes retenues de notre mémoire et Minamo, sont fx_me et fx_mo. Notons que la formule originale du calcul de probabilité n'est pas retenue car celle-ci est trop lente à l'exécution et n'améliore pas la convergence, en comparaison à nos deux adaptations.

Pour les six autres versions de *FiScIS-EA* (celles avec un comportement en dent de scie), les temps d'exécution sont, au contraire, plus lents si le nombre aléatoire q est tiré une unique fois (fx) pour tous les individus d'une même génération. De plus, les résultats sont moins bons, pour ces mêmes versions. Après avoir comparé les résultats de convergence et les graphiques de performance (en *SBO*), nous retenons pour le chapitre prochain DS_vr_or pour les exécutions avec l'AG pur ou intégré dans une boucle *SBO* et, en plus, DS_fx_or pour les cas tests résolus avec l'AG dans la boucle *SBO*.

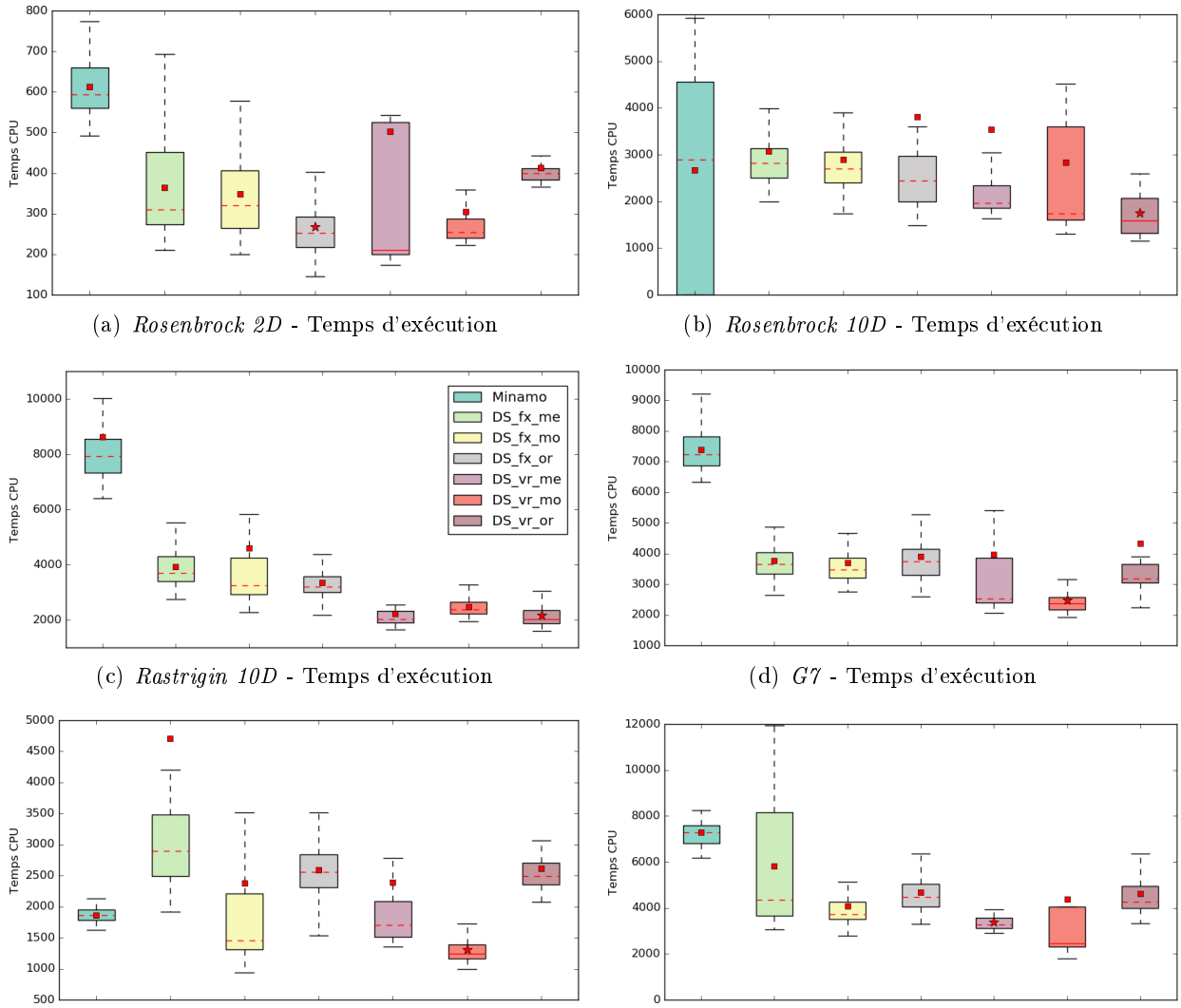


(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif (b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif (d) *G7* - Valeurs finales de la fonction objectif

(e) *G10* - Valeurs finales de la fonction objectif (f) *G2 plog 10* - Valeurs finales de la fonction objectif

FIGURE 6.7 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* en dent de scie et Minamo sur les six cas tests avec l'AG pur. Celles-ci concernent les valeurs finales des fonctions objectifs. La légende est la même pour toutes ces figures.



(a) *Rosenbrock 2D* - Temps d'exécution

(b) *Rosenbrock 10D* - Temps d'exécution

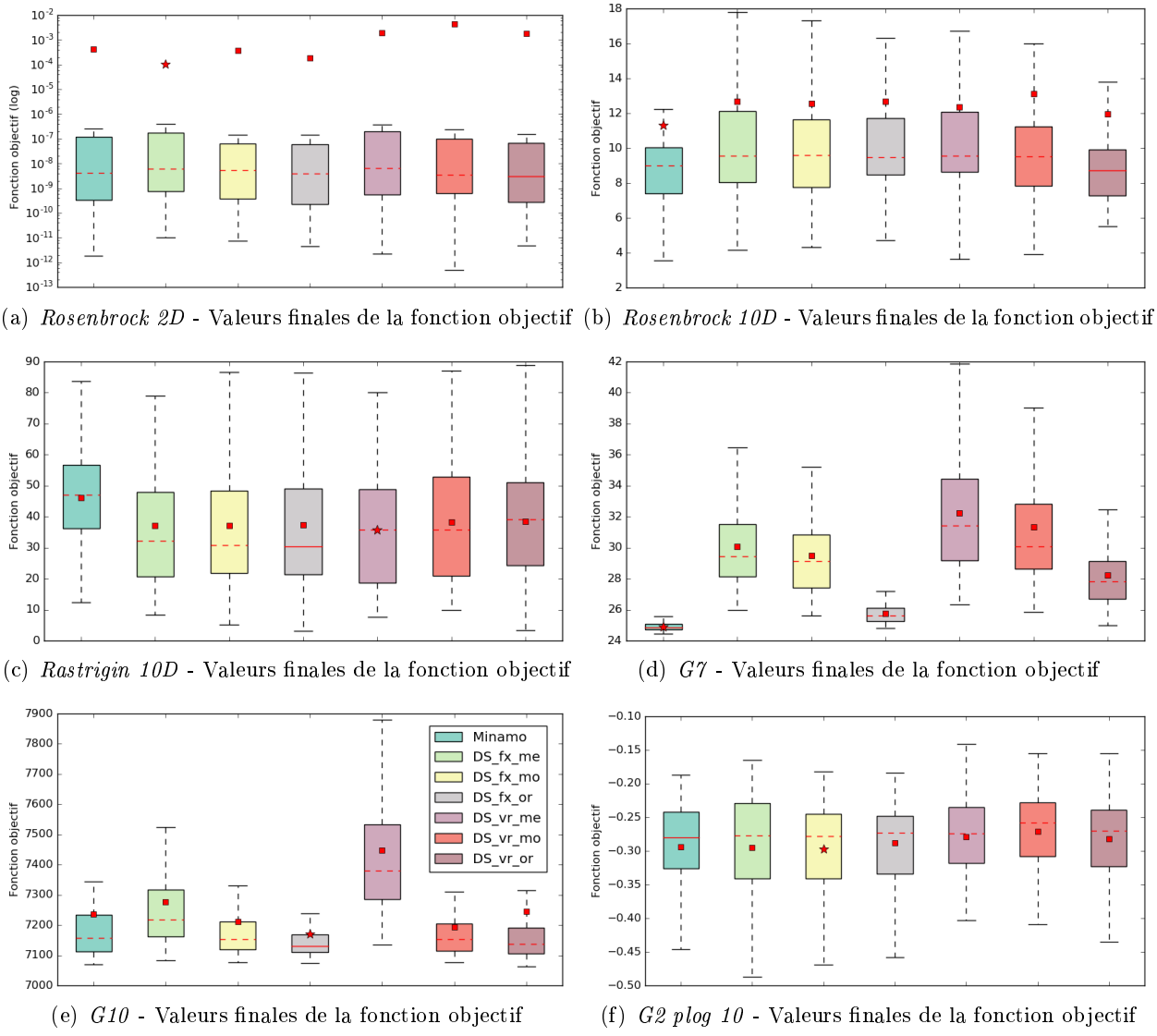
(c) *Rastrigin 10D* - Temps d'exécution

(d) *G7* - Temps d'exécution

(e) *G10* - Temps d'exécution

(f) *G2 plog 10* - Temps d'exécution

FIGURE 6.8 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* en dent de scie et Minamo sur les six cas tests avec l'AG pur. Celles-ci concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.

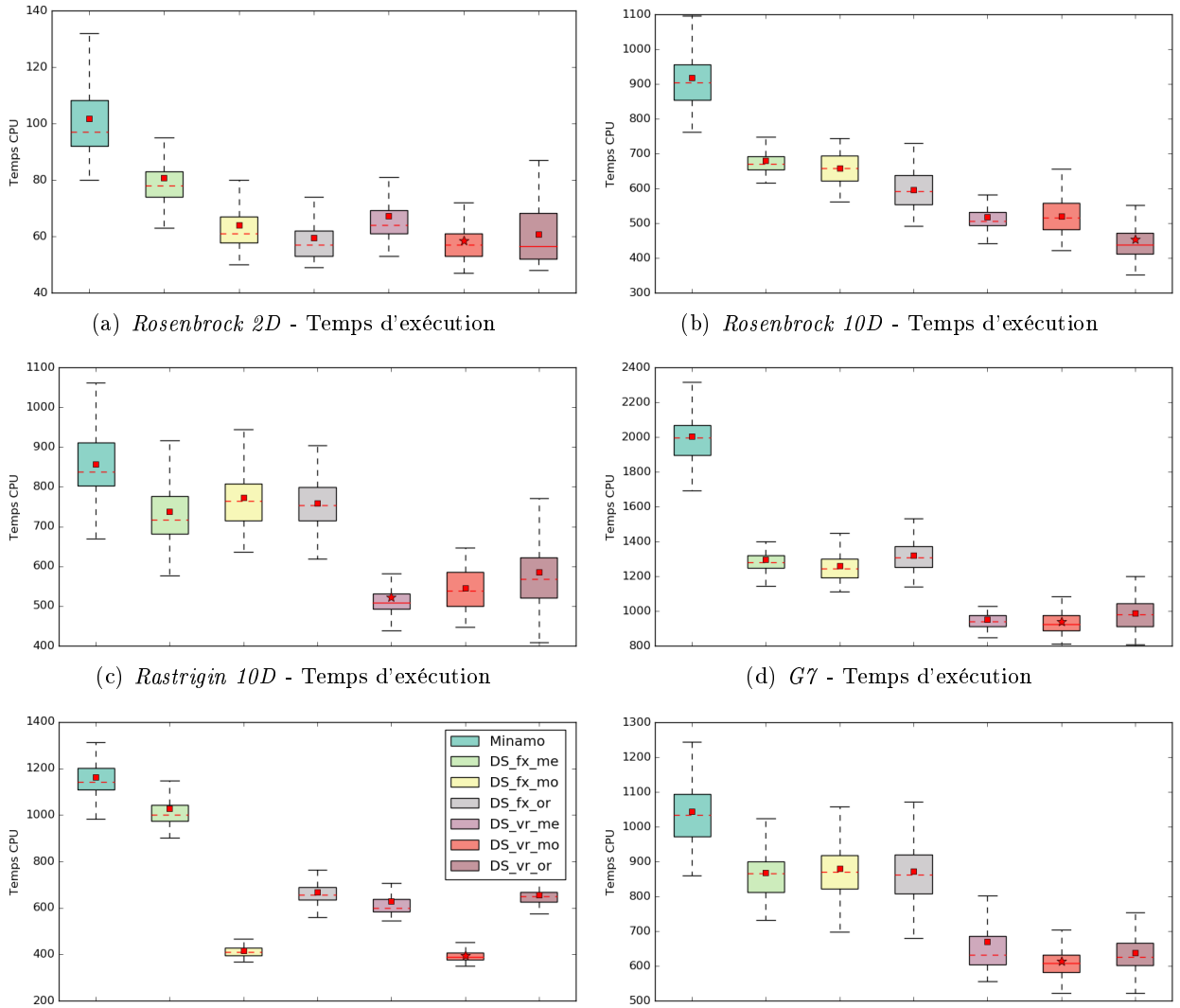


(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif (b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif (d) *G7* - Valeurs finales de la fonction objectif

(e) *G10* - Valeurs finales de la fonction objectif (f) *G2 plog 10* - Valeurs finales de la fonction objectif

FIGURE 6.9 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* en dent de scie et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent les valeurs finales des fonctions objectifs. La légende est la même pour toutes ces figures.



(a) *Rosenbrock 2D* - Temps d'exécution

(b) *Rosenbrock 10D* - Temps d'exécution

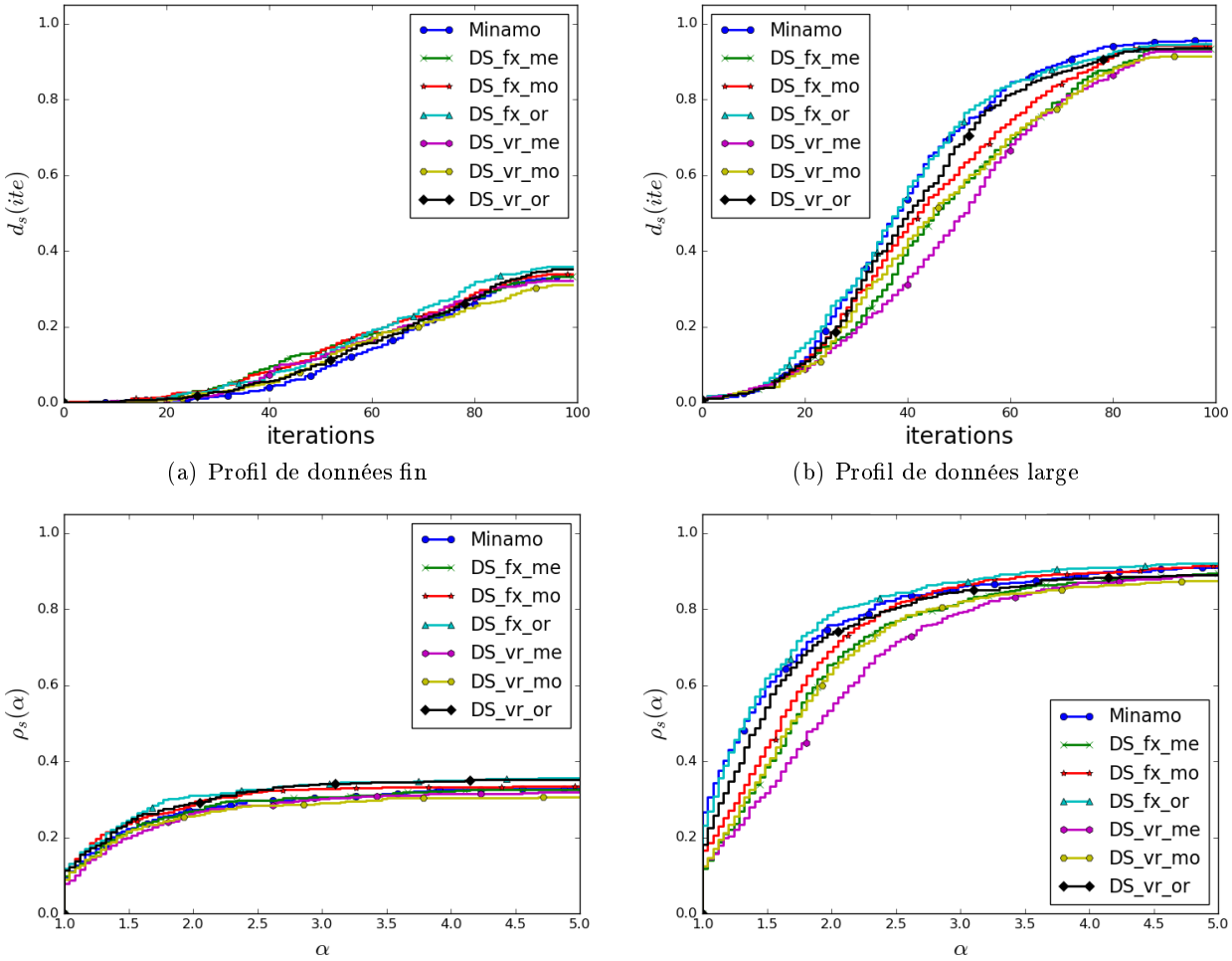
(c) *Rastrigin 10D* - Temps d'exécution

(d) *G7* - Temps d'exécution

(e) *G10* - Temps d'exécution

(f) *G2 plog 10* - Temps d'exécution

FIGURE 6.10 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions testées de *FiScIS-EA* en dent de scie et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.



(a) Profil de données fin

(b) Profil de données large

(c) Profil de performance fin

(d) Profil de performance large

FIGURE 6.11 – Les graphiques de profil de données et de performance sur les six cas tests pour les méthodes *FiScIS-EA* en dent de scie exécutées au sein de la boucle *SBO*.

Chapitre 7

Comparaison des meilleures méthodes

Nous allons commencer ce chapitre, par un rappel des méthodes que nous avons retenu des chapitres précédents. Ensuite, nous comparerons à nouveau ces méthodes entre elles et Minamo. D'abord, ceci sera réalisé sur les cas tests résolus avec l'AG pur. Puis, ceci sera fait sur ceux résolus avec l'AG en *SBO*. Le but de ce chapitre est de vérifier si les versions que nous avons introduites dans Minamo permettent d'améliorer ce code existant, en temps d'exécution et en convergence. Notons que nous aurions voulu réaliser ce chapitre sur plusieurs cas tests, autres que les six qui nous ont permis de sélectionner les méthodes. En effet, ceci aurait été encore plus intéressant, mais nous avons manqué de temps.

7.1 Rappel des versions retenues

Rappelons les méthodes retenues des chapitres précédents pour les exécutions de l'AG pur. Celles-ci sont :

- *Saw-Tooth* portant le tag NP_1 ($\overline{T} = 1$) ;
- version de *GAVaPS* BILIN_4_05 ;
- version de *GAVaPS* BILIN_4_06 ;
- version de *GAVaPS* PROP_4_05 ;
- version de *GAVaPS* PROP_4_06 ;
- *FiScIS-EA* avec l'adaptation fx_me ;
- *FiScIS-EA* avec l'adaptation fx_mo ;
- *FiScIS-EA* en dent de scie avec l'adaptation DS_vr_or.

La première version retenue est *Saw-Tooth* où $\overline{T}$ est le nombre de périodes de décroissance désirées durant les cent itérations de l'AG. Ensuite, nous avons retenu quatre versions de *GAVaPS*. Pour chacune d'elles, l'espérance de vie maximale EV_{max} est fixée à 4. La méthode BILIN_4_05 correspond au calcul bi-linéaire (BILIN) pour l'allocation de l'espérance de vie de chaque individu, où le taux de reproduction ρ vaut 0.5. Le calcul de l'espérance de vie est le même pour la version BILIN_4_06, mais ρ est fixé à 0.6. Pour PROP_4_05 et pour PROP_4_06, le calcul de l'espérance de vie est l'allocation proportionnelle (PROP), cependant PROP_4_05 utilise ρ qui vaut 0.5 et PROP_4_06 possède ρ qui est égal à 0.6. Concernant les versions de *FiScIS-EA* reprises dans ce chapitre, elles sont toutes adaptées. En effet, pour fx_me, *fx* signifie que nous tirons un nombre aléatoire q une unique fois (il est fixé) pour une génération donnée afin de le comparer aux probabilités de survie des individus et *me* représente le calcul médian que nous avons ajouté. De plus, *mo* représente le calcul moyen également introduit. Pour finir, DS_vr_or signifie que nous travaillons avec la méthode *FiScIS-EA* adaptée en dent de scie et utilisant sa formule originale (*or*) et tire un nombre aléatoire pour chaque individu (il est variable) afin d'être comparé avec la probabilité de survie. Toutes ces versions que nous avons implémentées dans Minamo seront comparées avec Minamo de base.

De plus, les méthodes retenues pour les exécutions de l'AG intégré au sein d'une boucle *SBO* sont :

— *Saw-Tooth* portant le tag NP_3 ($\overline{T} = 3$) ;
— *FiScIS-EA* avec l'adaptation fx_me ;
— *FiScIS-EA* avec l'adaptation fx_mo ;
— *FiScIS-EA* en dent de scie avec l'adaptation DS_vr_or ;
— *FiScIS-EA* en dent de scie avec l'adaptation DS_fx_or.

De même, toutes ces versions que nous avons implémentées dans Minamo seront comparées avec Minamo de base. Notons que pour l'AG intégré à la boucle *SBO*, aucune méthode provenant de *GAVaPS* ne sera comparée aux autres, car les méthodes telles qu'elles sont implémentées étaient trop lentes.

7.2 Comparaison pour l'AG pur

Dans les Figures 7.1 et 7.2, la valeur finale des fonctions objectifs et le nombre d'évaluations de fonctions selon les méthodes sont synthétisés pour les différents cas tests à partir de différentes exécutions de l'AG pur (au nombre de trente pour les méthodes de la famille *GAVaPS* et de cent pour les autres). Nous allons les analyser ici.

Tout d'abord, le nombre d'évaluations de fonctions nécessaires est réduit considérablement pour chacune de ces versions, comme le montrent ces deux figures. De plus, le temps d'exécution de ces différentes méthodes est également réduit par rapport à Minamo, comme le montre la Figure 15 de l'Annexe D.

Ensuite, l'unique version adaptée de *FiScIS-EA* en dent de scie (DS_vr_or) est la moins bonne de toutes au niveau de la convergence finale. En effet, pour tous les cas tests, les solutions obtenues sont plus élevées que les solutions des autres méthodes et même de Minamo. Par exemple, *Rastrigin 10D* (Figure 7.1(e)) et *G7* (Figure 7.2(a)) en sont de bonnes illustrations. Notons que cette adaptation n'est donc pas concluante en comparaison aux deux versions de *FiScIS-EA* (fx_me et fx_mo) qui utilisent la population d'enfants et des parents.

A côté de cela, nous constatons que les versions de *GAVaPS* (BILIN_4_05, BILIN_4_06, PROP_4_05 et PROP_4_06) détériorent bien souvent les solutions par rapport à Minamo. Le problème *G10* (Figure 7.2(c)) est l'illustration la plus typique. Cependant, les versions dans lesquelles l'espérance de vie des individus est attribuée bi-linéairement (BILIN_4_05 et BILIN_4_06) améliorent la solution finale par rapport à Minamo pour *Rosenbrock 2D* (Figure 7.1(a)) et ne la détériorent pas pour *Rastrigin 10D*. De plus, les versions pour lesquelles l'espérance de vie des individus est attribuée proportionnellement (PROP_4_05 et PROP_4_06) ne détériorent pas la recherche de solution pour *Rastrigin 10D*.

Par contre, la meilleure méthode est sans aucun doute *Saw-Tooth* qui ne possède qu'une dent (NP_1), c'est-à-dire qu'une décroissance linéaire de la population. En partant d'une population contenant assez d'individus, il est simplement intéressant de décroître linéairement la taille de cette population, car cela offre de meilleures solutions en un temps divisé par deux, comme le montrent ces six cas tests (Figures 7.1 et 7.2).

De plus, les versions améliorées de *FiScIS-EA* (fx_me et fx_mo) obtiennent de très bons résultats aussi. En effet, les solutions finales obtenues par ces méthodes pour tous les cas tests (sauf pour *Rosenbrock 10D*, selon la Figure 7.1(c)) sont meilleures que celles renvoyées par Minamo.

Concernant le nombre d'évaluations de fonctions, nous pouvons remarquer qu'il n'y a pas de lien entre la solution finale obtenue pour une méthode et le nombre d'évaluations de fonctions. En effet, la version DS_vr_or nécessite plus d'évaluations de fonctions que les autres méthodes (hors Minamo) et elle possède les moins bons résultats (par exemple, le problème *G7* avec la Figure 7.2(a) et le problème *G2 plog 10* avec la Figure 7.2(e)). Cependant, pour ces deux mêmes problèmes, les versions de *GAVaPS*, qui possèdent le moins d'évaluations de fonctions (BILIN_4_05 et PROP_4_05), possèdent également de mauvaises convergences.

Dans les Figures 7.3 et 7.4, nous représentons l'évolution de la taille de la population déterminée par les huit méthodes retenues et Minamo, sur les six cas tests. Nous ne pouvions pas calculer une moyenne de l'évolution de cette taille à partir des trente ou cent exécutions différentes pour chaque méthode. En effet, les décroissances ou les augmentations de la taille de la population ne sont pas réalisées durant les mêmes itérations sur des exécutions différentes, comme un léger décalage de quelques itérations apparaît souvent. Ainsi, l'évolution de la population représentée dans ces figures provient de la meilleure exécution de chacune des méthodes. De plus, pour mieux visualiser les courbes, nous présentons cinq méthodes par graphe. Dans ces deux pages, les courbes de gauche concernent les méthodes de *GAVaPS* (BILIN_4_05, BILIN_4_06, PROP_4_05 et PROP_4_06) et celles de droite concernent la méthode de *Saw-Tooth* à une période (NP_1) et les trois méthodes de *FiScIS-EA* (fx_me, fx_mo et DS_vr_or). A chaque fois, la taille fixe de Minamo est reprise.

Dans ces figures (Figures 7.3 et 7.4), les méthodes de *GAVaPS* ne font pas assez évoluer la taille de la population au cours des itérations (courbes de gauche), en comparaison aux méthodes de *FiScIS-EA* (courbes de droite). En effet, la taille de la population déterminée par les méthodes de *GAVaPS* demeure constante après une vingtaine d'itérations pour les problèmes sans contraintes : *Rosenbrock 2D* (Figure 7.3(a)), *Rosenbrock 10D* (Figure 7.3(c)) et *Rastrigin 10D* (Figure 7.3(e)). Cependant, deux oscillations sont induites par ces versions de *GAVaPS* sur les problèmes avec contraintes *G7* (Figure 7.4(a)) et *G10* (Figure 7.4(c)). En comparaison, la taille de la population déterminée par les méthodes de *FiScIS-EA* varie d'avantage au cours des itérations (voir les courbes de droite de ces deux figures). Pour les six cas tests, les stratégies de *FiScIS-EA* déterminent par elles-mêmes une taille plus élevée au début des itérations. En effet, un pic de la taille de la population est observable pour tous ces problèmes. Ceci n'est pas le cas pour les exécutions de *GAVaPS*, comme nous venons de le souligner. De plus, des pics supplémentaires lors des exécutions de *FiScIS-EA* apparaissent durant les itérations. Ceci leur confère un avantage par rapport à *GAVaPS*.

De plus, nous avons dit précédemment que les versions de *GAVaPS*, qui possèdent le moins d'évaluations de fonctions (BILIN_4_05 et PROP_4_05), possèdent également de mauvaises convergences. Ces mauvais résultats pour *G2 plog 10* (Figure 7.4(e)) peuvent être expliqués également par le fait que la taille de la population n'a pas assez augmentée au début des itérations. Ainsi, cela démontre l'idée que nous avions soulevée dans le résumé du chapitre concernant *GAVaPS* (Section 5.5), qui pour rappel avait été suggérée que Cook et Tauritz [3].

Concernant l'évolution de la taille de la population déterminée par la version *FiScIS-EA* à caractère de dent de scie (DS_vr_or), celle-ci s'adapte pour chaque problème d'optimisation. En effet, pour les problèmes de *Rosenbrock* en 2D et en 10D (Figure 7.3(b) et 7.3(d)) peu d'oscillations sont réalisées. Par contre, pour le problème *Rastrigin 10D* (Figure 7.3(f)) qui possède énormément de minima locaux, les oscillations sont espacées de quelques itérations seulement. Pourtant, cela n'est pas synonyme de bons résultats, comme nous l'avons écrit précédemment (Figure 7.1(e)). Par contre, cette méthode en dent de scie (DS_vr_or) a décidé de ne réaliser qu'une seule oscillation pour le problème *G10* (Figure 7.4(d)), comme *Saw-Tooth* (NP_1). Un tel comportement a induit des résultats comparables à Minamo (Figure 7.2(c)). Ceci démontre également la puissance de *Saw-Tooth* à une oscillation.

Le temps d'exécution est, quant à lui, souvent amélioré par rapport à Minamo, comme le montre la Figure 15 de l'Annexe D. En effet, le temps d'exécution de toutes les versions exécutées sur *Rastrigin 10D* (Figure 15(c)), *G7* (Figure 15(d)) et *G2 plog 10* (Figure 15(f)) est réduit par rapport à Minamo. Cependant, pour les problèmes *Rosenbrock 10D* (15(b)) et *G10* (Figure 15(e)), les temps d'exécution fluctuent autour du temps d'exécution de Minamo. De plus, comme nous l'avons déjà dit, le temps d'exécution de la version adaptée en dent de scie de *FiScIS-EA* (DS_vr_or) est la plus lente des versions nouvellement implémentées dans Minamo, pour les problèmes avec contraintes. Remarquons également que le temps d'exécution de la version *Saw-Tooth* (NP_1) est plus étendu que Minamo pour *Rosenbrock 2D* (Figure 15(a)) et plus élevé (en plus d'être plus étendu) pour *G10* (Figure 15(e)). Cependant, comme nous l'avons expliqué dans la Section 4.3 du Chapitre 4, nous pensons que ces temps d'exécution ont été atteints par un évènement externe à nos exécutions.

Pour conclure, si nous devons conseiller une unique méthode à garder pour les exécutions avec l'AG pur ce serait *Saw-Tooth* avec une unique décroissance linéaire (NP_1). En effet, même si pour certains cas tests cette version semble plus lente, cela ne semblait pas être causé par celle-ci (voir l'explication dans la Section 4.3 du Chapitre 4). Nous sommes convaincus que cette méthode diminue le nombre d'évaluations de fonctions de moitié pour chaque exécution et permet d'obtenir de très bons résultats, même meilleurs que Minamo. Ceci est vraiment bénéfique. De plus, les versions de *FiScIS-EA* adaptées (fx_me et fx_mo) doivent également être gardées. En effet, celles-ci fournissent de très bons résultats et nous sommes sûrs que le temps d'exécution est réduit, comme le nombre d'évaluations de fonctions est également réduit. Notons que les résultats de *GAVaPS* ne sont pas concluant car la taille initiale de la population (fixée à 20 individus) n'induit pas un assez bon signal.

7.3 Comparaison pour l'AG intégré dans la boucle *SBO*

Dans les Figures 7.5 et 7.6, nous avons repris la convergence finale et le temps d'exécution des différentes méthodes retenues pour l'AG intégré à la boucle *SBO*. Nous allons les analyser.

D'emblée, nous constatons que les méthodes adaptées de *FiScIS-EA* (fx_me et fx_mo) ont une convergence équivalente à Minamo pour tous les cas tests, hormis le problème sans contraintes *Rastrigin 10D* (Figure 7.5(e)). Cependant, celles-ci ne permettent pas toujours de réduire le temps d'exécution. En effet, bien que la durée d'exécution soit réduite pour *Rosenbrock 2D* (Figure 7.5(b)) et *G7* (Figure 7.6(b)) ou constante pour *Rosenbrock 10D* (Figure 7.5(d)) et *G10* (Figure 7.6(d)) par rapport à Minamo, le temps est augmenté pour *Rastrigin 10D* (Figure 7.5(f)) et *G2 plog 10* (Figure 7.6(f)).

Concernant l'unique version de *Saw-Tooth* (NP_3), dans laquelle le nombre de périodes est ici fixé à 3, la convergence finale est semblable à Minamo pour tous les cas tests, sauf *G7* (Figure 7.6(a)). Cependant le temps d'exécution est toujours réduit, il est divisé par deux pour les problèmes avec contraintes (*G7*, *G10* et *G2 plog 10*).

Les versions adaptées de *FiScIS-EA* en dent de scie (DS_vr_or et DS_fx_or) renvoient toujours des solutions semblables à celles que renvoie Minamo, en un temps réduit ! Cependant, pour le problème *G7* (Figure 7.6(a)), la valeur finale est détériorée par rapport à Minamo. Par contre, ce problème est résolu en un temps d'exécution amélioré. La durée d'exécution de ces versions est assez proche de celle de NP_3. Par contre, DS_vr_or est plus rapide pour les problèmes *Rosenbrock 10D* (Figure 7.5(d)) et *G7* (Figure 7.6(b)), mais la solution finale est moins bonne que Minamo. De plus, pour ces mêmes problèmes, la durée d'exécution de la méthode DS_fx_or est plus lente que NP_3.

Pour mieux comparer la convergence et la performance de toutes ces méthodes exécutées en *SBO*, nous avons également réalisé les profils de données et de performance dans la Figure 7.7. D'abord, pour un seuil fin, le pourcentage de solutions trouvées est amélioré pour toutes ces méthodes par rapport à Minamo (voir Figure 7.7(a)) et la performance est également améliorée par rapport à Minamo pour toutes ces méthodes (voir Figure 7.7(c)). De plus, pour un seuil large, la version de *Saw-Tooth* où le nombre de périodes est fixé à 3 est équivalente à Minamo pour le profil de données (Figure 7.7(b)) et pour la robustesse (Figure 7.7(d)). En plus de ceci, nous constatons que les méthodes de *FiScIS-EA* adaptées (fx_me et fx_mo) et la version de *Saw-Tooth* (NP_3) ont des profils de données semblables pour un seuil fin (Figure 7.7(a)) et des profils de performance semblables pour un seuil fin (voir Figure 7.7(c)). Ces trois versions sont également plus performantes que les versions de *FiScIS-EA* adaptées en dent de scie (DS_vr_or et DS_fx_or) où ces dernières sont équivalentes entre elles pour la performance et les solutions finales.

En conclusion, si nous devons retenir trois méthodes, celles-ci seraient également la version de *Saw-Tooth*, où cette fois-ci le nombre de périodes est fixé à 3 (NP_3), et les deux méthodes adaptées de *FiScIS-EA* (fx_me et fx_mo), en prenant garde que ces deux dernières versions sont parfois plus lentes.

7.4 En résumé

Dans ce chapitre, nous avons réalisé un comparatif des méthodes que nous avons retenues au fil de l'avancement de notre mémoire. Nous en retenons trois parmi celles-ci pour les exécutions de l'AG pur et trois autres pour les exécutions de l'AG intégré dans une boucle *SBO*. Pour l'AG pur, les versions les plus performantes qui permettent donc une amélioration de la recherche de minima et une réduction du temps d'exécution sont :

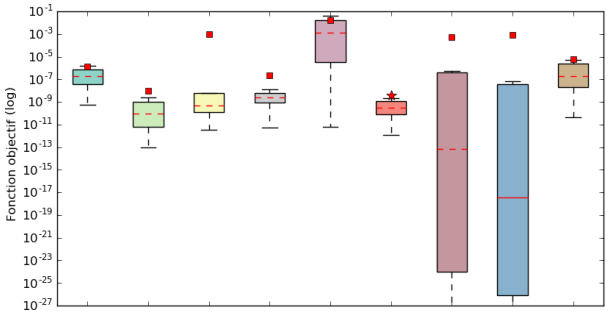
- NP_1 (version de *Saw-Tooth* avec le paramètre $\overline{T}$ valant 1) ;
- fx_me (version améliorée de *FiScIS-EA* où le calcul de la probabilité de survie pour chaque individu est réalisé par une formule de médiane et où le nombre aléatoire q qui est comparé avec la probabilité de chaque individu n'est tirée qu'une unique fois pour une génération donnée) ;
- fx_mo (version améliorée de *FiScIS-EA* utilisant une formule de moyenne et où q est également tiré une unique fois par génération).

De plus, pour l'AG intégré dans une boucle *SBO*, les trois meilleures versions pour la convergence sont :

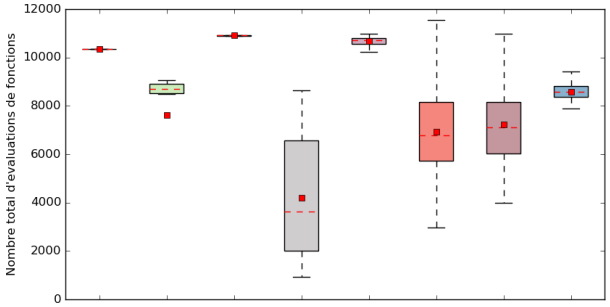
- NP_3 (version de *Saw-Tooth* avec le paramètre $\overline{T}$ valant 3) ;
- fx_me ;
- fx_mo.

Notons que les deux versions de *FiScIS-EA* ne permettent pas d'avoir pour tous les cas tests une réduction du temps de calcul.

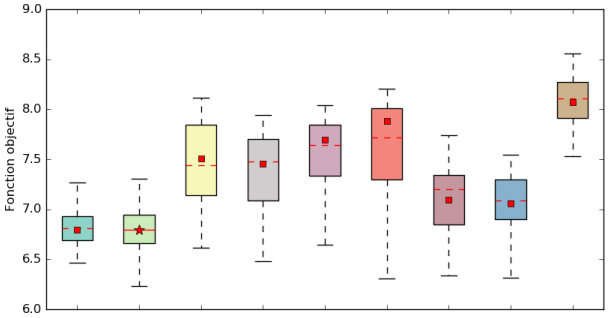
Remarquons que, pour pouvoir affirmer que ces versions à garder sont bel-et-bien les meilleures parmi celles retenues, nous devrons prochainement les exécuter sur plusieurs autres cas tests et réaliser à nouveau des profils de performance et de données.



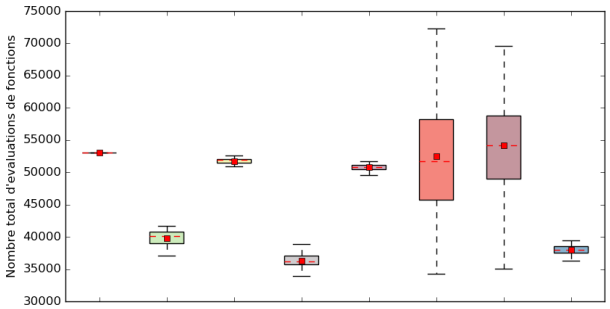
(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif



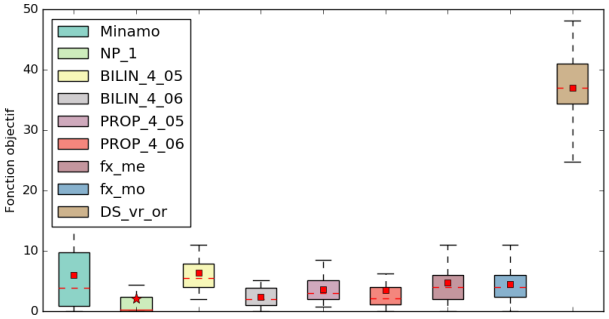
(b) *Rosenbrock 2D* - Nombre d'évaluations de fonctions (Minamo : 20'000 évaluations)



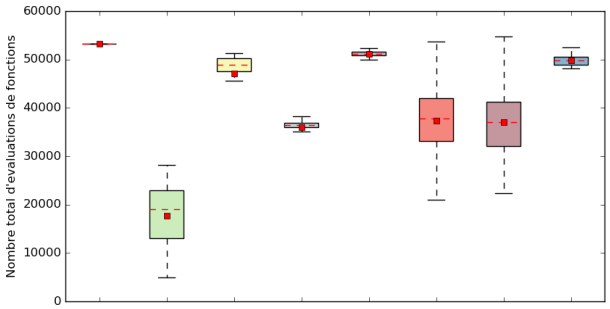
(c) *Rosenbrock 10D* - Valeurs finales de la fonction objectif



(d) *Rosenbrock 10D* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

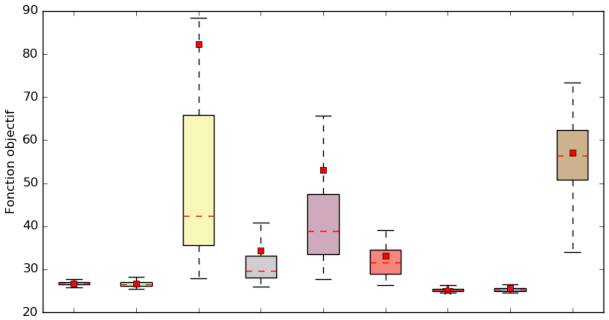


(e) *Rastrigin 10D* - Valeurs finales de la fonction objectif

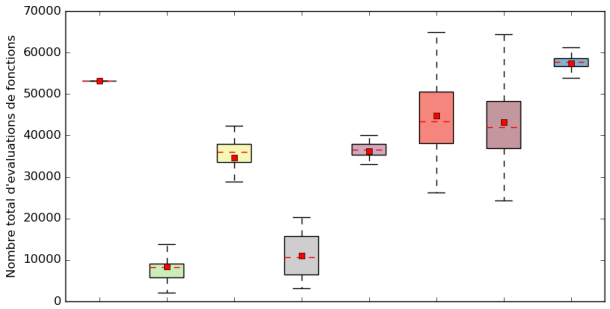


(f) *Rastrigin 10D* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

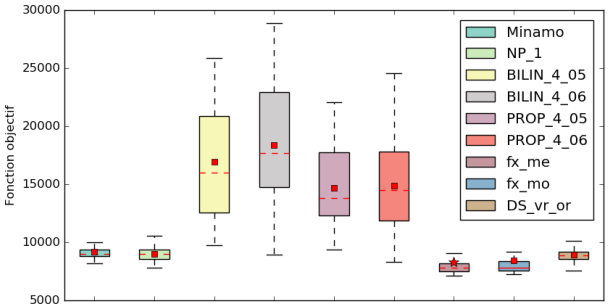
FIGURE 7.1 – Ces boîtes à moustache représentent les résultats synthétiques des différentes méthodes retenues durant le mémoire et Minamo sur les trois problèmes sans contraintes avec l'AG pur. Celles-ci concernent les valeurs finales des fonctions objectifs, à gauche, et le nombre d'évaluations de fonctions à droite. Le nombre d'évaluations de fonctions réalisées par Minamo pour chaque méthode est repris dans la description de chaque problème. La légende est la même pour toutes ces figures.



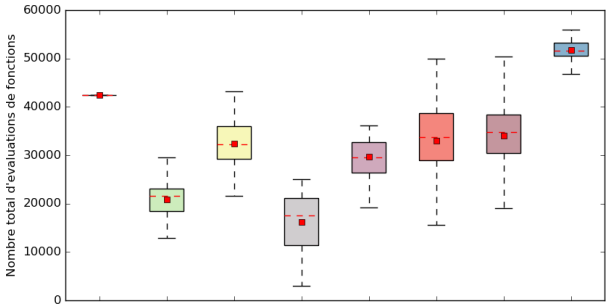
(a) *G7* - Valeurs finales de la fonction objectif



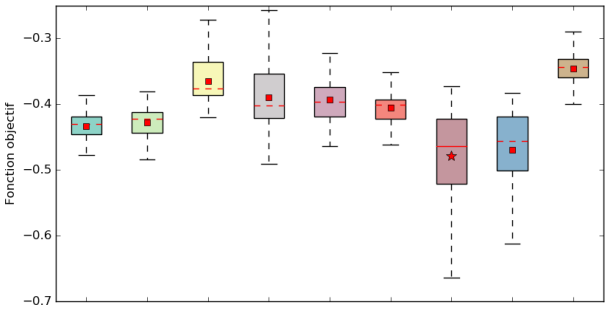
(b) *G7* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)



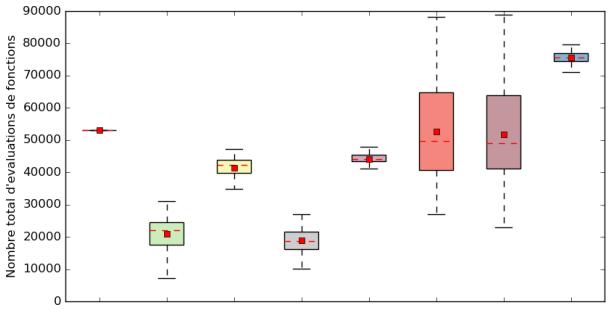
(c) *G10* - Valeurs finales de la fonction objectif



(d) *G10* - Nombre d'évaluations de fonctions (Minamo : 80'000 évaluations)

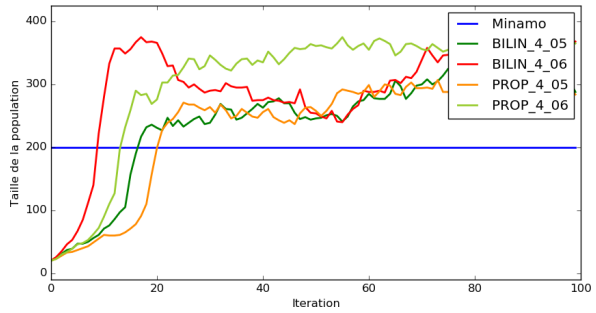


(e) *G2 plog 10* - Valeurs finales de la fonction objectif

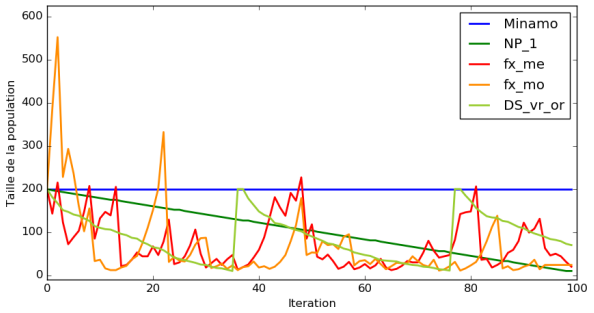


(f) *G2 plog 10* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

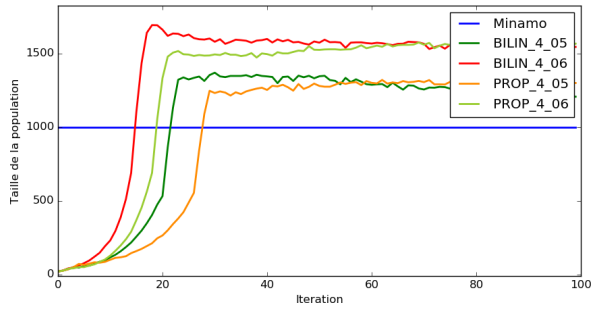
FIGURE 7.2 – Ces boîtes à moustache représentent les résultats synthétiques des différentes méthodes retenues durant le mémoire et Minamo sur les trois problèmes sans contraintes avec l'AG pur. Celles-ci concernent les valeurs finales des fonctions objectifs, à gauche, et le nombre d'évaluations de fonctions à droite. Le nombre d'évaluations de fonctions réalisées par Minamo pour chaque méthode est repris dans la description de chaque problème. La légende est la même pour toutes ces figures.



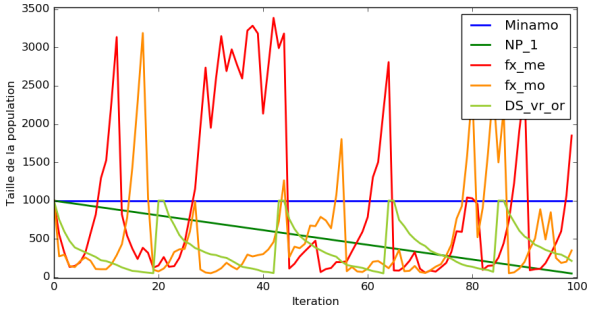
(a) *Rosenbrock 2D* - Évolution de la taille de la population (méthodes *GAVaPS*)



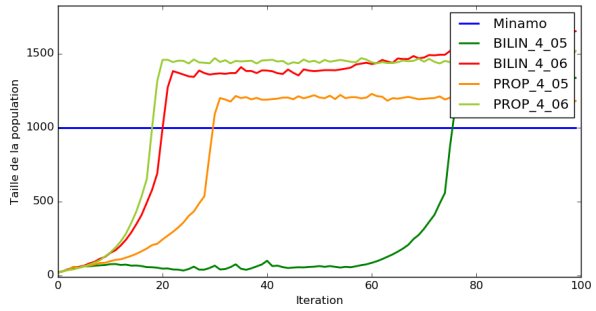
(b) *Rosenbrock 2D* - Évolution de la taille de la population (méthodes *Saw-Tooth* et *FiScIS-EA*)



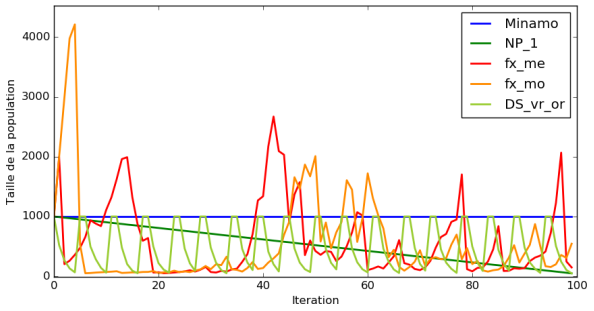
(c) *Rosenbrock 10D* - Évolution de la taille de la population (méthodes *GAVaPS*)



(d) *Rosenbrock 10D* - Évolution de la taille de la population (méthodes *Saw-Tooth* et *FiScIS-EA*)

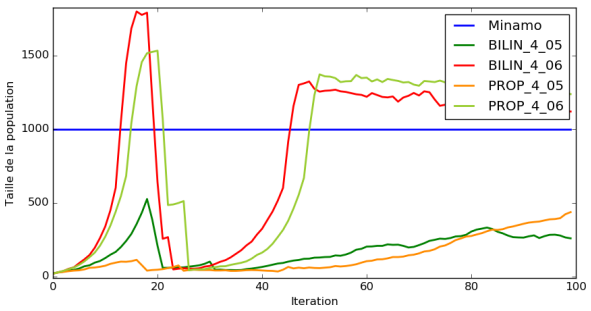


(e) *Rastrigin 10D* - Évolution de la taille de la population (méthodes *GAVaPS*)

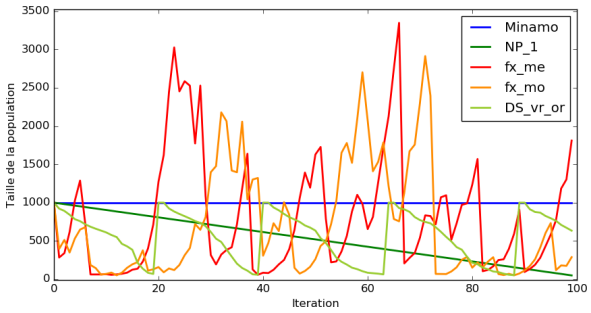


(f) *Rastrigin 10D* - Évolution de la taille de la population (méthodes *Saw-Tooth* et *FiScIS-EA*)

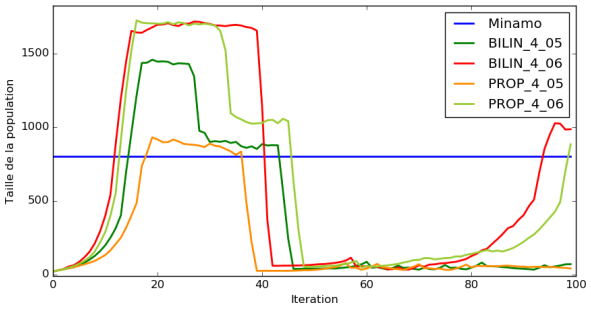
FIGURE 7.3 – Ces graphiques représentent l'évolution de la taille de la population au cours des itérations de la meilleure exécution de chaque méthode retenue, sur les problèmes sans contraintes. A gauche, ces courbes concernent les quatre méthodes de *GAVaPS* et à droite celles-ci concernent la méthode *Saw-Tooth* et les trois méthodes *FiScIS-EA*.



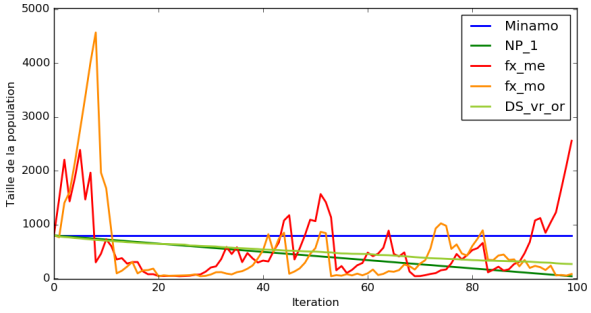
(a) *G7* - Évolution de la taille de la population (méthodes *GAVaPS*)



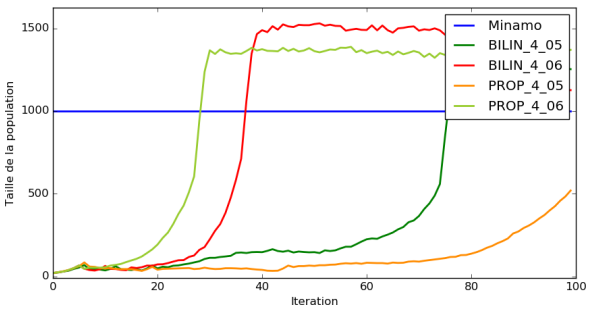
(b) *G7* - Évolution de la taille de la population (méthodes *Saw-Tooth* et *FiScIS-EA*)



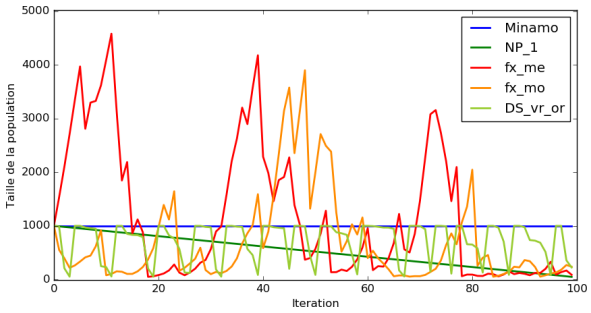
(c) *G10* - Évolution de la taille de la population (méthodes *GAVaPS*)



(d) *G10* - Évolution de la taille de la population (méthodes *Saw-Tooth* et *FiScIS-EA*)

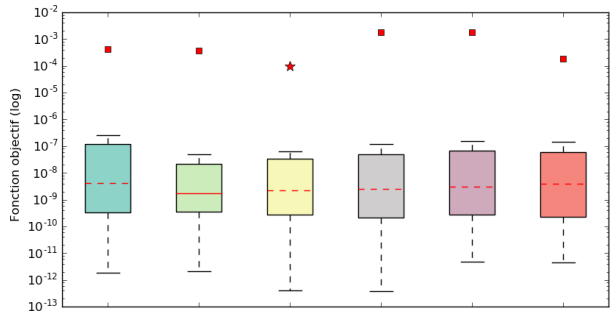


(e) *G2 plog 10* - Évolution de la taille de la population (méthodes *GAVaPS*)

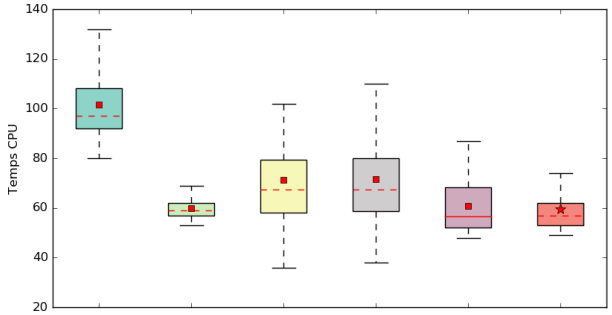


(f) *G2 plog 10* - Évolution de la taille de la population (méthodes *Saw-Tooth* et *FiScIS-EA*)

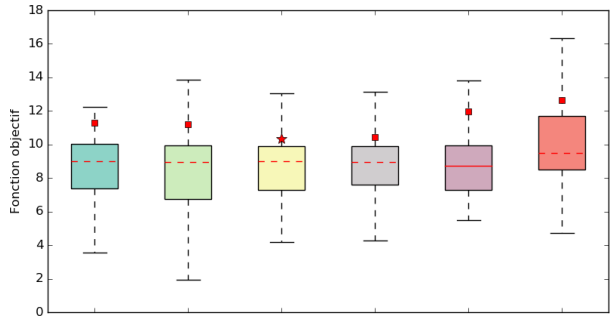
FIGURE 7.4 – Ces graphiques représentent l'évolution de la taille de la population au cours des itérations de la meilleure exécution de chaque méthode retenue, sur les problèmes avec contraintes. A gauche, ces courbes concernent les quatre méthodes de *GAVaPS* et à droite celles-ci concernent la méthode *Saw-Tooth* et les trois méthodes *FiScIS-EA*.



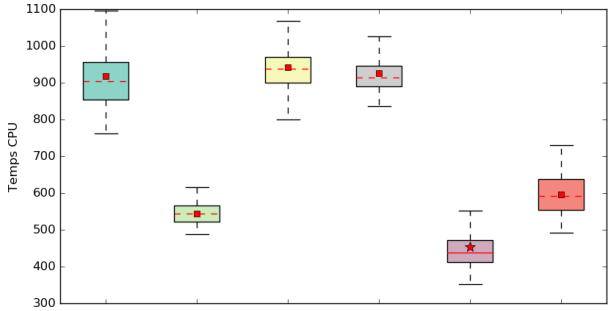
(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif



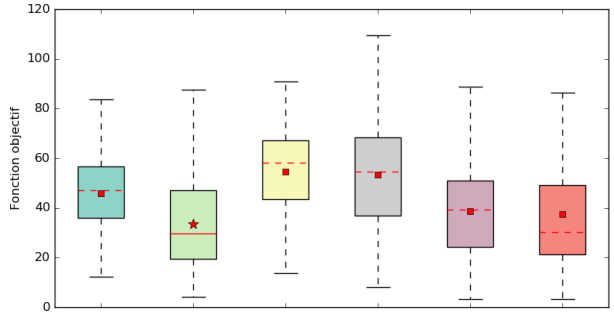
(b) *Rosenbrock 2D* - Temps d'exécution



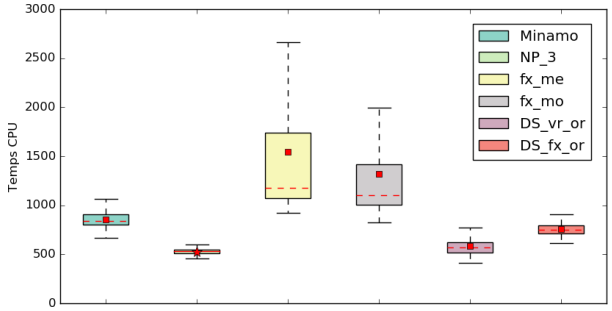
(c) *Rosenbrock 10D* - Valeurs finales de la fonction objectif



(d) *Rosenbrock 10D* - Temps d'exécution

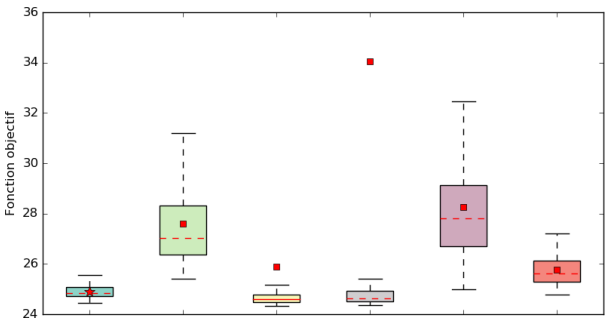


(e) *Rastrigin 10D* - Valeurs finales de la fonction objectif

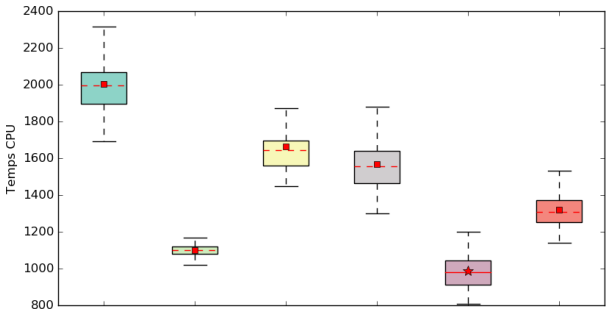


(f) *Rastrigin 10D* - Temps d'exécution

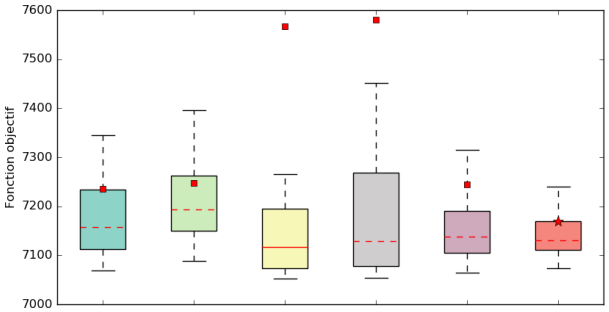
FIGURE 7.5 – Ces boîtes à moustache représentent les résultats synthétiques des différentes méthodes retenues durant le mémoire et Minamo sur les cas tests sans contraintes résolus avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent les valeurs finales des fonctions objectifs, à gauche, et le temps d'exécution en seconde à droite. La légende est la même pour toutes ces figures.



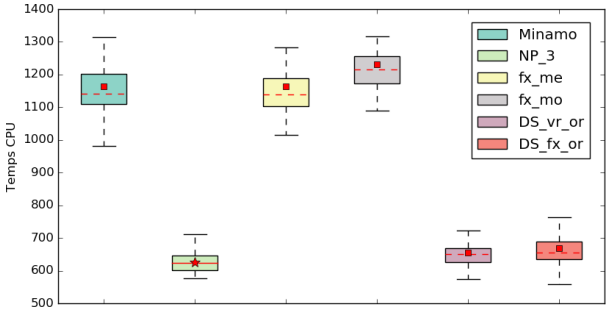
(a) *G7* - Valeurs finales de la fonction objectif



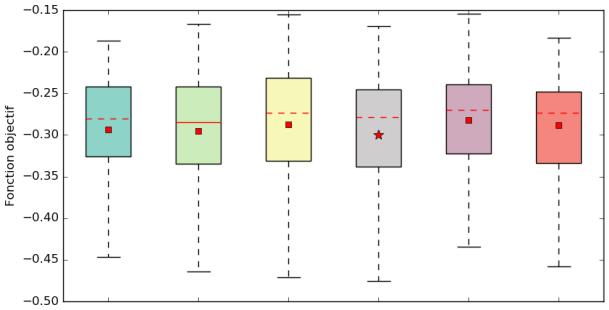
(b) *G7* - Temps d'exécution



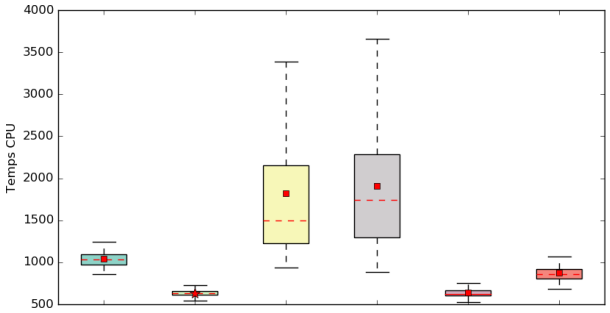
(c) *G10* - Valeurs finales de la fonction objectif



(d) *G10* - Temps d'exécution

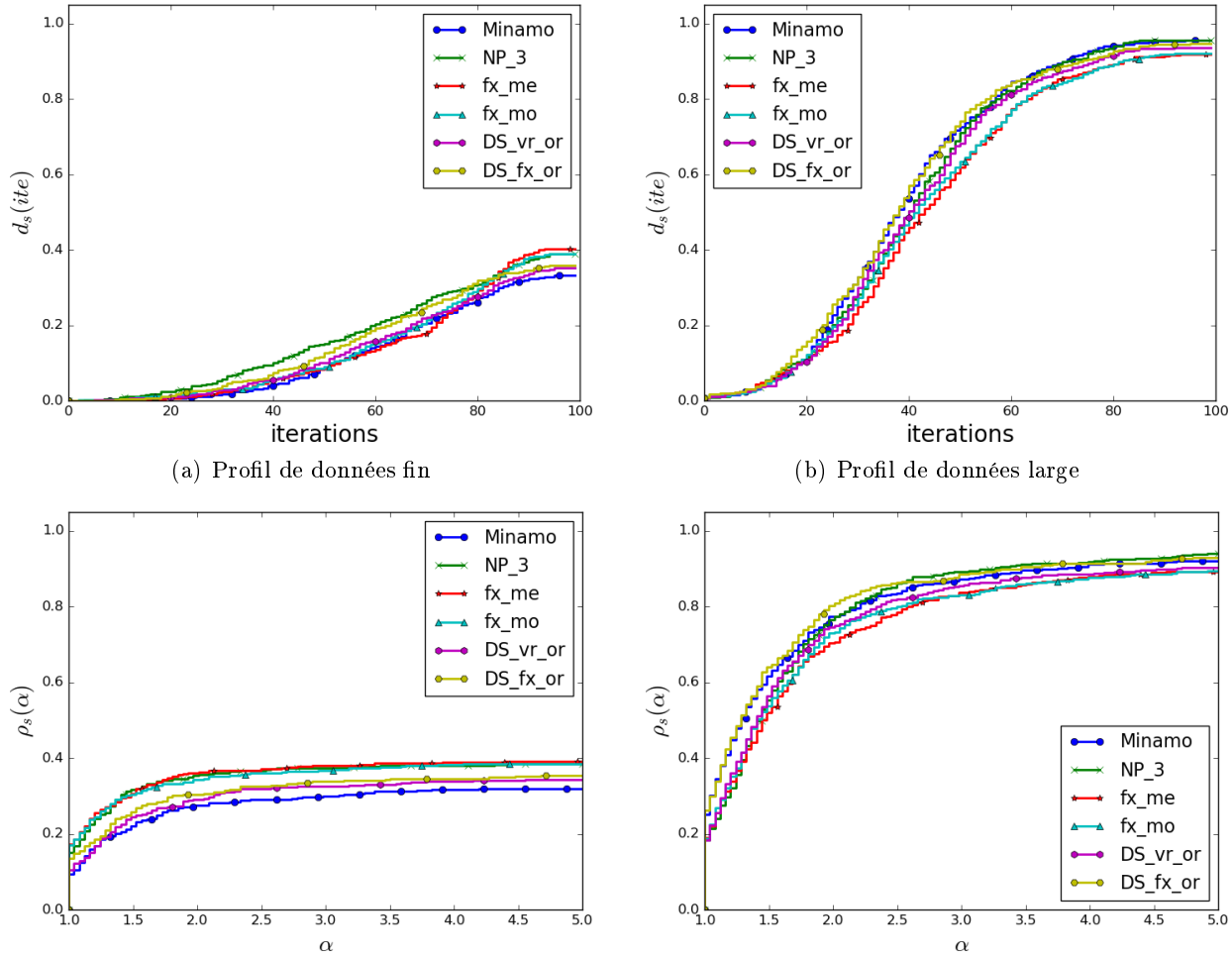


(e) *G2 plog 10* - Valeurs finales de la fonction objectif



(f) *G2 plog 10* - Temps d'exécution

FIGURE 7.6 – Ces boîtes à moustache représentent les résultats synthétiques des différentes méthodes retenues durant le mémoire et Minamo sur les cas tests avec contraintes résolus avec l'AG intégré dans une boucle *SBO*. Celles-ci concernent les valeurs finales des fonctions objectifs, à gauche, et le temps d'exécution en seconde à droite. La légende est la même pour toutes ces figures.



(a) Profil de données fin

(b) Profil de données large

(c) Profil de performance fin

(d) Profil de performance large

FIGURE 7.7 – Les graphiques de profil de données et de performance sur les six cas tests pour les différentes méthodes retenues au cours de ce mémoire exécutées au sein de la boucle *SBO*.

Conclusion et réflexions futures

Le but de ce mémoire était de contrôler, au fil des itérations, la taille de la population μ d'un algorithme génétique (AG) qui peut être intégré dans une optimisation assistée par modèles de substitution. L'espoir était d'améliorer la vitesse de convergence sans détériorer la recherche de solution par rapport au programme d'optimisation multi-disciplinaire (Minamo) développé à Cenaero.

Pour ce faire, dans le Chapitre 1, nous avons posé le cadre de notre recherche, en rappelant quelques définitions. Puis, nous avons décrit l'AG avec ses différentes subtilités (ses paramètres et ses opérateurs). Dans le Chapitre 2, nous avons fait l'état de l'art de différents réglages sur les paramètres de l'AG, sur des combinaisons de paramètres et sur ses opérateurs. Ensuite, dans le Chapitre 3, nous avons présenté l'AG de Minamo, avec ses paramètres et ses opérateurs. L'avantage de ce programme est l'utilisation d'un méta-modèle (boucle *SBO*) qui approxime la fonction objectif et sur lequel l'AG est appliqué. Ce méta-modèle évolue durant les itérations de la boucle *SBO*. Nous avons donc travaillé avec l'AG pur (sans le méta-modèle) et avec l'AG intégré au sein d'une boucle *SBO*. Après cette présentation, nous avons réalisé trois chapitres pour contrôler la taille de la population μ au cours des itérations au sein de l'AG de Minamo. Le but des approches qui ont été introduites dans Minamo était d'améliorer la rapidité de Minamo sans détériorer la recherche de solution. D'abord, la méthode *Saw-Tooh* proposée par Koumousis et Katsaras [1] a été développée (Chapitre 4) : celle-ci décroît linéairement la taille de la population durant plusieurs itérations puis elle réinitialise la population à sa taille originale. Ensuite, l'algorithme *GAVaPS* d'Arabas et al. [2] dont l'acronyme signifie *Genetic Algorithm with Varying Population Size* (Chapitre 5) a été établi dans Minamo avec plusieurs adaptations. Cette approche calcule l'espérance de vie pour chaque individu créé lors de la recherche ; celle-ci est le nombre d'itérations durant lesquels l'individu doit rester. La dernière méthode développée a été *FiScIS-EA* de Cook et Tauritz [3] signifiant *Fitness Scaled Individual Survival Evolutionary Algorithm* (Chapitre 6). Cette démarche calcule pour chaque individu une probabilité de survie pour être introduit dans la génération suivante. Pour cette méthode, nous avons introduit des variantes en s'inspirant des deux premières démarches. Pour finir, dans le Chapitre 7, nous avons comparé différentes versions retenues au cours de ces trois chapitres afin d'en garder trois pour l'AG pur et trois pour l'AG intégré dans la boucle *SBO*.

Deux méthodes (*GAVaPS* et *FiScIS-EA*) nécessitaient d'injecter les enfants dans la population de parents avant de la réduire pour l'itération suivante. Nous avons dû implémenter cette idée dans Minamo. Cependant, la réduction de la population n'étant pas assez restrictive en utilisant les formules de ces méthodes, nous avons introduit une condition supplémentaire afin de ne pas ajouter les individus qui font régresser la recherche.

Pour comparer ces méthodes avec Minamo, ou entre elles, nous avons ajouté un axe d'analyse, souvent absent des articles scientifiques : le temps d'exécution. De fait, les publications ne répertorient bien souvent que le nombre d'évaluations de fonctions. Or, nous l'avons constaté, même si ce nombre diminue de moitié, le temps d'exécution n'est pas diminuer de moitié. Cela est plus remarqué pour les problèmes résolus avec l'AG au sein d'une boucle *SBO*. En effet, les méthodes introduites utilisent également du temps de calcul pour gérer la population et, lors d'évaluations rapides de fonctions (comme lors de l'évaluation de fonctions sur un méta-modèle), le gain est moindre.

Ainsi, de la manière dont nous avons implémenté la méthode *GAVaPS*, le temps d'exécution, en utilisant l'AG au sein d'une boucle *SBO*, était bien souvent empiré par rapport à Minamo. Cependant, pour l'AG pur, nous avons pu faire ressortir quelques versions de *GAVaPS*. Ces versions concernaient la manière de calculer l'espérance de vie, l'espérance de vie maximale EV_{max} et le taux de reproduction ρ (différent de p_c), en utilisant l'allocation bi-linaire ou proportionnelle. Les combinaisons étaient les suivantes : $EV_{max} = 4$ et $\rho = 0.5$; $EV_{max} = 4$ et $\rho = 0.6$; $EV_{max} = 5$ et $\rho = 0.6$.

Pour *Saw-Tooth*, la réduction de moitié du nombre d'évaluations de fonctions est véritablement bénéfique pour les exécutions de l'AG pur et pour ceux de l'AG intégré au sein d'une boucle *SBO*, en terme de temps de calcul. De plus, une amélioration des solutions finales est également notable lorsqu'une unique période de réduction de la population est réalisée (pour l'AG pur) ou quand trois périodes de réduction de la population sont réalisées (pour l'AG intégré à une boucle *SBO*).

La méthode *FiScIS-EA* calcule la probabilité de survie par une formule appelée originale (*or*) et tire un nombre aléatoire q pour chaque individu d'une génération (*vr*). Pour chaque individu, le nombre est alors comparé avec la probabilité de survie qui lui est associée afin de déterminer s'il peut rester pour la génération suivante. Nous avons apporté deux modifications. Premièrement, comme le calcul original ressemble à la fraction de l'allocation linéaire de *GAVaPS*, nous avons modifié ce calcul en s'inspirant de l'allocation bi-linaire de *GAVaPS* : nous avons utilisé une formulation avec la moyenne des valeurs de fitness d'une génération (*mo*) et une formulation avec la médiane (*me*). Deuxièmement, le nombre q pouvait être tiré une unique fois pour une génération donnée (*fx*).

Pour les problèmes résolus avec l'AG pur, la version originale de *FiScIS-EA* (combinant *vr* et *or*) est moins performante que les autres méthodes nouvellement introduites. En effet, le temps d'exécution était plus lent par rapport à nos méthodes, comme cette méthode évaluait d'avantage les fonctions. Notons que, par rapport à Minamo, toutes ces versions de *FiScIS-EA* permettaient de réduire le temps de calcul et de garder des solutions finales semblables, voire d'obtenir de meilleures solutions. Les meilleurs d'entre elles étaient : la combinaison de *fx* et *mo* et la combinaison de *fx* et *me* (en écartant donc le calcul original de *FiScIS-EA*). Celles-ci sont également les meilleures versions testées de *FiScIS-EA* avec des adaptations, pour les problèmes résolus avec l'AG au sein d'une boucle *SBO*. En effet, la recherche n'est pas détériorée par rapport à Minamo. Cependant, le temps de calcul est parfois détérioré pour certains cas tests.

Une autre adaptation de *FiScIS-EA* a été réalisée. Au lieu d'ajouter les enfants à la population de parents, comme le fait la méthode originale, seuls les enfants sont pris pour remplacer la population. Ainsi le nombre d'individus décroît et, lorsque la taille de la population a atteint la taille minimale, la population est augmentée aléatoirement avec des individus. Cette adaptation a introduit un caractère de dent de scie à la méthode. Cependant, la convergence était détériorée par rapport à Minamo, aussi bien pour l'AG pur ou pour celui-ci intégré dans une boucle *SBO*.

Pour conclure, nous devons retenir la méthode *Saw-Tooth* où une unique décroissance linéaire de la taille de la population est utilisée pour les résolutions de l'AG pur et maintenir celle-ci avec trois décroissances suivies d'une réinitialisation de la population à sa taille originale. En effet, ces méthodes améliorent la convergence par rapport à Minamo et réduisent le temps d'exécution.

Prochainement, nous allons tester ces méthodes sur plusieurs autres problèmes d'optimisation afin de vérifier nos constats en utilisant des profils de données et de performance. Nous utiliserons également des tests statistiques afin de s'assurer que les résultats ne sont pas détériorés ou sont améliorés. De plus, comme nous avons contrôlé uniquement la taille de la population au sein de l'AG, d'autres adaptations pour d'autres paramètres peuvent faire le sujet de nouvelles études.

Bibliographie

[1] V. Koumousis et C. Katsaras, A Saw-tooth Genetic Algorithm Combining the Effects of Variable Population Size and Reinitialization to Enhance Performance, *IEEE Transactions on Evolutionary Computation*, vol. 10, p. 19–28, 2006.

[2] J. Arabas, Z. Michalewicz et J. Mulawka, GAVaPS - a Genetic Algorithm with Varying Population Size, *Proceedings of the First IEEE Conference on Evolutionary Computation*, p. 73–78, 1994.

[3] J. Cook et D. Tauritz, An Exploration into Dynamic Population Sizing, *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, p. 807–814, 2010.

[4] C. Blum et A. Roli, Metaheuristics in Combinatorial Optimization : Overview and Conceptual Comparison, *ACM Computing Surveys (CSUR)*, vol. 35, p. 268–308, 2003.

[5] K. Socha et M. Dorigo, Ant Colony Optimization for Continuous Domains, *European Journal of Operational Research*, vol. 185, p. 1155–1173, 2008.

[6] A. Eiben, Z. Michalewicz, M. Schoenauer et J. Smith, Parameter Control in Evolutionary Algorithms, *Springer Verlag*, vol. 54, p. 19–46, 2007.

[7] B. Miller et D. Goldberg, Genetic Algorithms, Tournament Selection, and the Effects of Noise, *Complex systems*, vol. 9, p. 193–212, 1995.

[8] T. Pencheva, K. Atanassov et A. Shannon, Modelling of a Roulette Wheel Selection Operator in Genetic Algorithms Using Generalized Nets, *International Journal of Bioautomation*, vol. 13, p. 257–264, 2009.

[9] J. Holland, Adaptation in Natural and Artificial Systems : an Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, *International Journal of Bioautomation*, 1992.

[10] GEATbx, Evolutionary Algorithms 3 Selection, `http://www.geatbx.com/docu/algindex-02.html`, consulté le vendredi 18 mai 2018, 2006.

[11] K. Deb, An Efficient Constraint Handling Method for Genetic Algorithms, *Computer Methods in Applied Mechanics and Engineering*, vol. 186, p. 311–338, 2000.

[12] D. Wolpert et W. Macready, No Free Lunch Theorems for Optimization, *IEEE transactions on Evolutionary Computation*, vol. 1, p. 67–82, 1997.

[13] E. Pellerin, L. Pigeon et S. Delisle, Self-Adaptive Parameters in Genetic Algorithms, *Data Mining and Knowledge Discovery - DATAMINE*, vol. 5433, p. 53–64, 2004.

[14] G. Karafotias, M. Hoogendoorn et A. Eiben, Parameter Control in Evolutionary Algorithms : Trends and Challenges, *IEEE Transactions on Evolutionary Computation*, vol. 19, p. 167–187, 2015.

[15] G. Harik et F. Lobo, A parameter-less Genetic Algorithm, *Illinois Genetic Algorithms Laboratory*, vol. 1, 1999.

[16] E. Smorodkina et D. Tauritz, Greedy Population Sizing for Evolutionary Algorithms, *Proceedings of the 2007 IEEE Congress on Evolutionary Computation*, p. 2181–2187, 2007.

[17] R. Hinterding, Z. Michalewicz et T. Peachey, Self-adaptive Genetic Algorithms for Numeric Functions, *Parallel Problem Solving from Nature*, vol. 4, p. 420–429, 1996.

[18] R. Smith et E. Smuda, Adaptively Resizing Populations : Algorithm, Analysis and First Results, *Complex Systems*, vol. 9, p. 47–72, 1995.

[19] T.-L. Yu, K. Sastry et D. Goldberg, On-line Population Size Adjusting Using Noise and Substructural Measurements, *Illinois Genetic Algorithms Laboratory*, vol. 3, 2005.

[20] A. Eiben, Z. Michalewicz, M. Schoenauer et J. Smith, Is Self-adaptation of Selection Pressure and Population Size Possible ? - A Case Study, *Lecture Notes in Computer Science*, vol. 9, p. 900–909, 2006.

[21] J. Teo, Exploring Dynamic Self-adaptive Populations in Differential Evolution, *Soft Computing*, vol. 10, p. 673–686, 2006.

[22] T. Back et M. Schutz, Intelligent Mutation Rate Control in Canonical Genetic Algorithms, *International Symposium on Methodologies for Intelligent Systems*, vol. 13, p. 158–167, 1996.

[23] T. Bäck, A. Eiben et N. van der Vaart, An Empirical Study on GAs "without Parameters", *Parallel Problem Solving from Nature*, vol. 6, p. 315–324, 1994.

[24] D. Goldberg, A note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing, *Complex Systems*, vol. 4, p. 445–460, 1990.

[25] M. Affenzeller, Segregative Genetic Algorithms (SEGA) : A Hybrid Superstructure Upwards Compatible to Genetic Algorithms for Retarding Premature Convergence, *International Journal Comput. Syst. Signal*, vol. 2, p. 16–30, 2001.

[26] A. Kaveh et M. Shahrouzi, Dynamic Selective Pressure Using Hybrid Evolutionary and Ant System Strategies for Structural Optimization, *International Journal for Numerical Methods in Engineering*, vol. 73, p. 544–563, 2008.

[27] J. Hesser et R. Männer, Towards an Optimal Mutation Probability for Genetic Algorithms, *Proceedings of the 1st Conference on Parallel Problem Solving from Nature*, 1990.

[28] M. Srinivas et L. Patnaik, Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms, *IEEE Transactions on Systems, Man and Cybernetic*, vol. 24, p. 656–667, 1994.

[29] Y.-Y. Wong, K.-H. Lee, K.-S. Leung et C.-W. Ho, A Novel Approach in Parameter Adaptation and Diversity Maintenance for Genetic Algorithms, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, vol. 7, p. 506–515, 2003.

[30] A. Aleti et I. Moser, Predictive Parameter Control, *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, p. 561–568, 2011.

[31] M. Maruo, H. Lopes et M. Delgado, Self-Adapting Evolutionary Parameters : Encoding Aspects for Combinatorial Optimization Problems, *Evolutionary Computation in Combinatorial Optimization*, 2005.

[32] O. Kramer, Evolutionary Self-adaptation : a Survey of Operators and Strategy Parameters, *Evolutionary Intelligence*, vol. 3, p. 51–65, 2010.

[33] S. Meyer-Nieberg et H.-G. Beyer, Self-adaptation in Evolutionary Algorithms, *Parameter Setting in Evolutionary Algorithms*, p. 47–76, 2007.

[34] B. Mc Ginley, J. Maher, C. O' Riordan et F. Morgan, Maintaining Healthy Population Diversity Using Adaptive Crossover, Mutation and Selection, *IEEE Transactions on Evolutionary Computation*, vol. 15, p. 692–714, 2011.

[35] E. Smorodkina et D. Tauritz, Toward Automating EA Configuration : The Parent Selection Stage, *Proceedings of the 2007 IEEE Congress on Evolutionary Computation*, p. 63–70, 2007.

[36] L. Davis, Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.

[37] F. Herrera et M. Lozano, Adaptation of Genetic Algorithm Parameters Based on Fuzzy Logic Controllers, *Genetic Algorithms and Soft Computing*, vol. 8, p. 95–125, 1996.

[38] M. Lee et H. Takagi, Dynamic Control of Genetic Algorithms Using Fuzzy Logic Techniques, *Proceedings of the Fifth International Conference on Genetic Algorithms*, p. 76–83, 1993.

[39] Cenaero, *Minamo 3.0.4 - Theoretical Manual*, 2018.

[40] C. Sainvitu, V. Iliopoulou et I. Lepot, Global Optimization with Expensive Functions - Sample Turbomachinery Design Application, *Recent Advances in Optimization and its Applications in Engineering, Springer*, p. 499–509, 2010.

[41] C. Bishop, *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

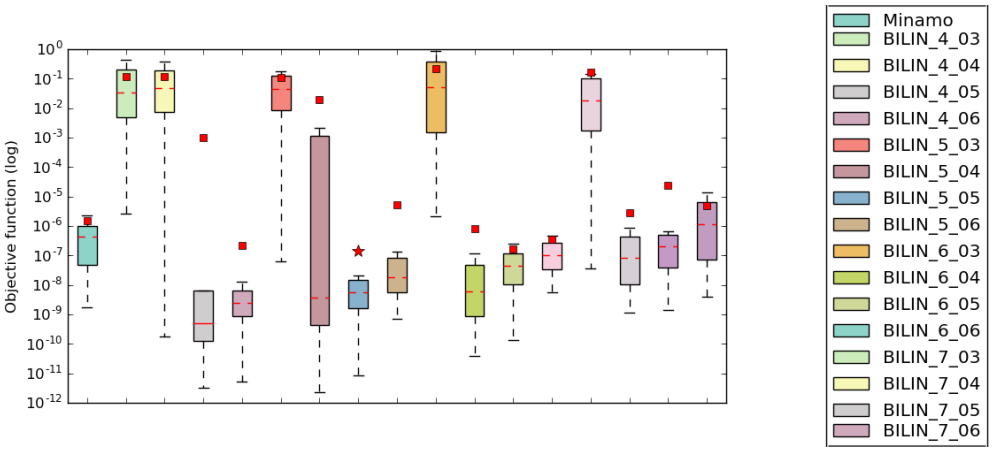
[42] Y.-W. Shang et Y.-H. Qiu, A Note on the Extended Rosenbrock Function, *IEEE Evolutionary Computation*, vol. 14, p. 119–126, 2006.

[43] R. G. Regis et C. A. Shoemaker, A Quasi-multistart Frameworkfor Global Optimization of Eexpensive Functions using Response Surface Models, *Journal of Global Optimization*, vol. 56, p. 1719–1753, 2012.

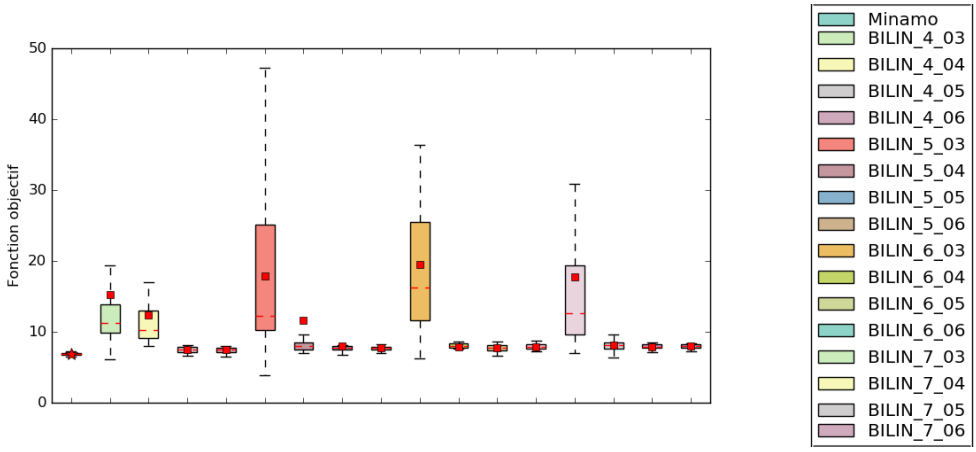
[44] J. More et S. Wild, Benchmarking Derivative-free Optimization Algorithms, *SIAM Journal on Optimization*, vol. 20, p. 172–191, 2009.

Annexes

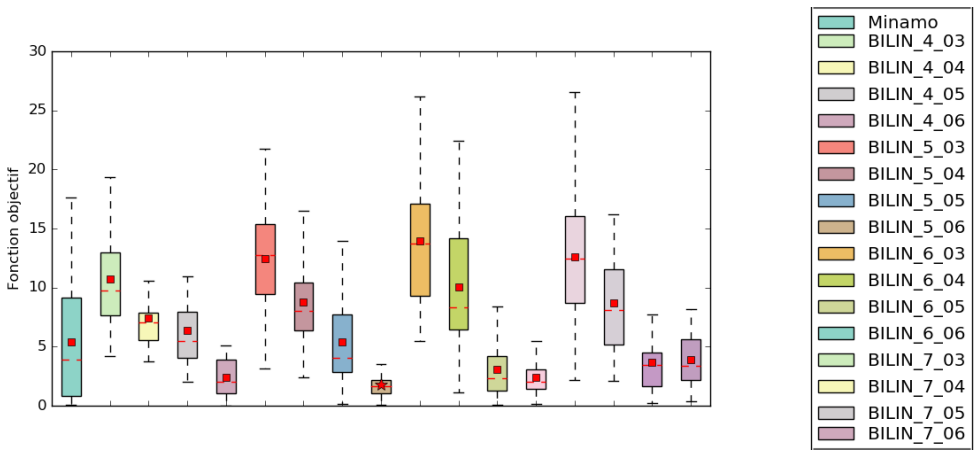
Annexe A. *GAVaPS*



(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif

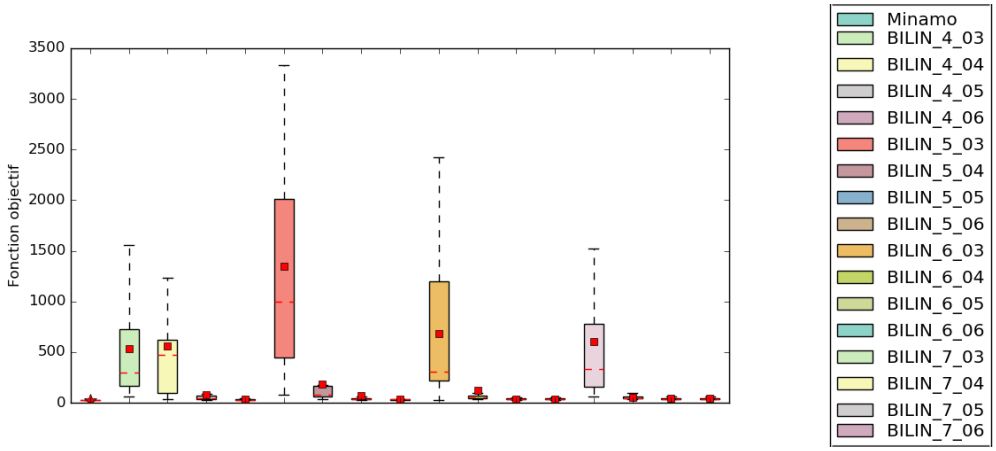


(b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

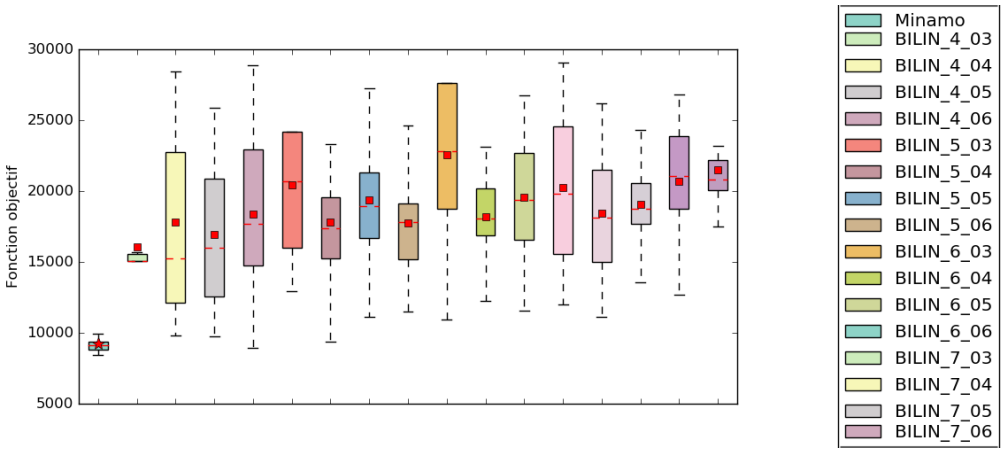


(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif

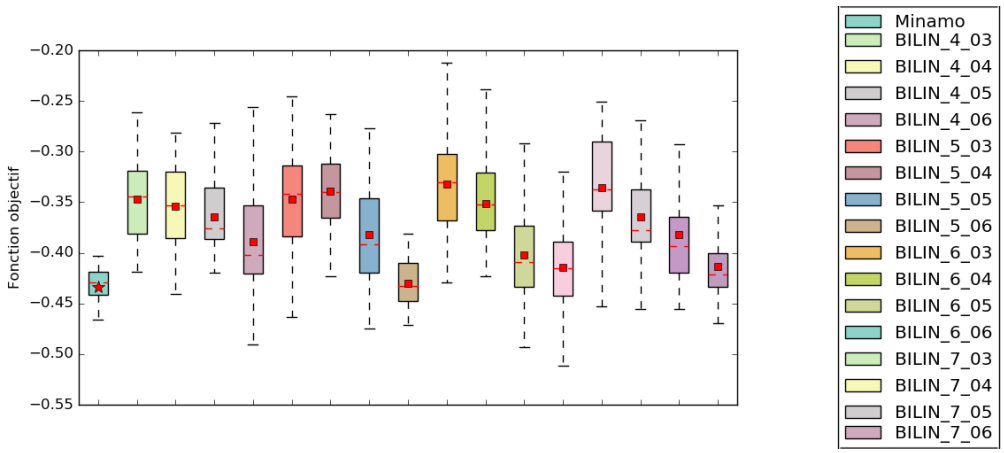
FIGURE 1 – Ces graphiques représentent les valeurs finales des différentes stratégies *GAVaPS* contenant le calcul bi-linéaire, qui ont été exécutées avec l'AG pur sur les différents problèmes sans contraintes.



(a) *G7* - Valeurs finales de la fonction objectif

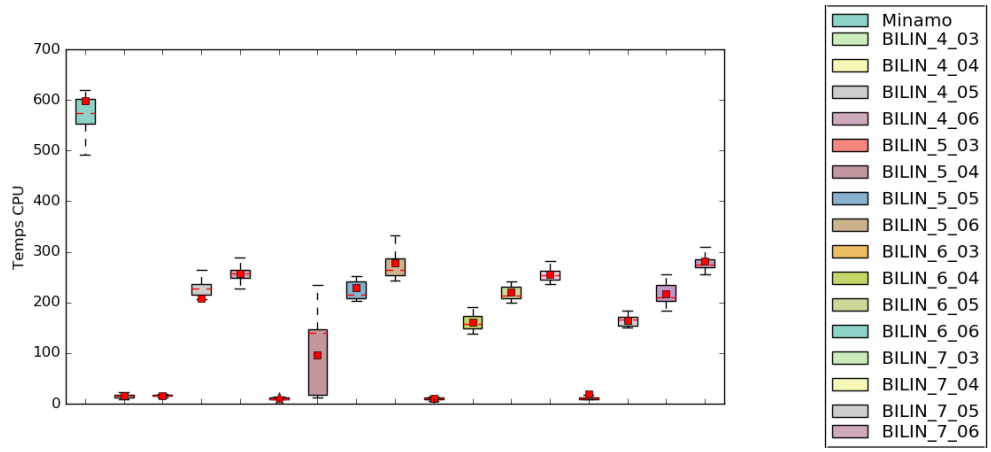


(b) *G10* - Valeurs finales de la fonction objectif

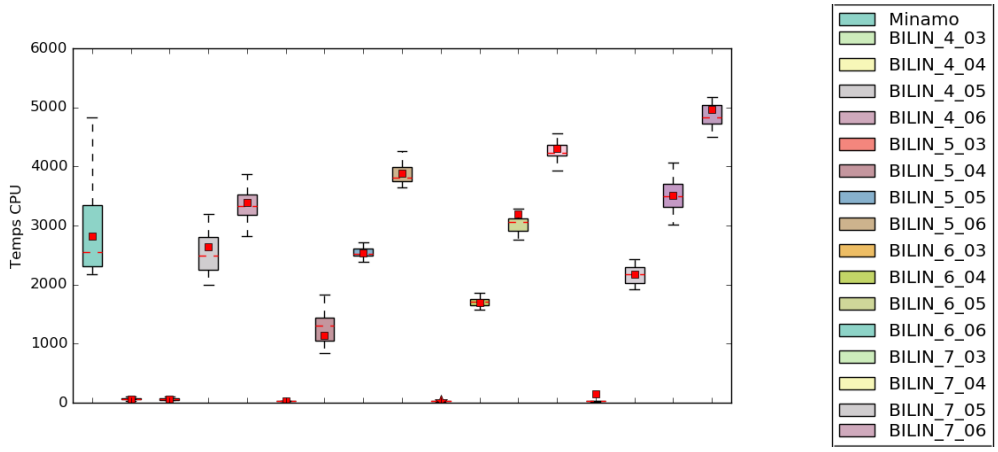


(c) *G2 plog 10* - Valeurs finales de la fonction objectif

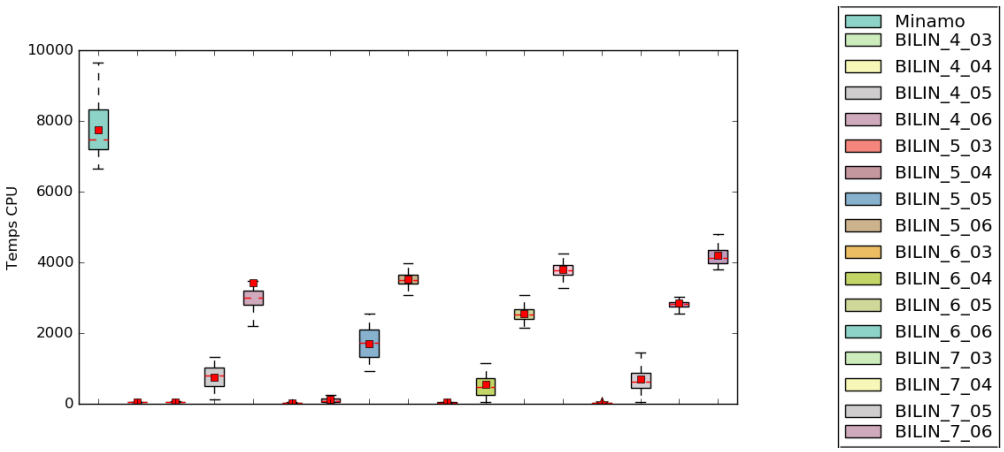
FIGURE 2 – Ces graphiques représentent les valeurs finales des différentes stratégies *GAVaPS* contenant le calcul bi-linéaire, qui ont été exécutées avec l'AG pur sur les différents problèmes avec contraintes.



(a) *Rosenbrock 2D* - Temps d'exécution

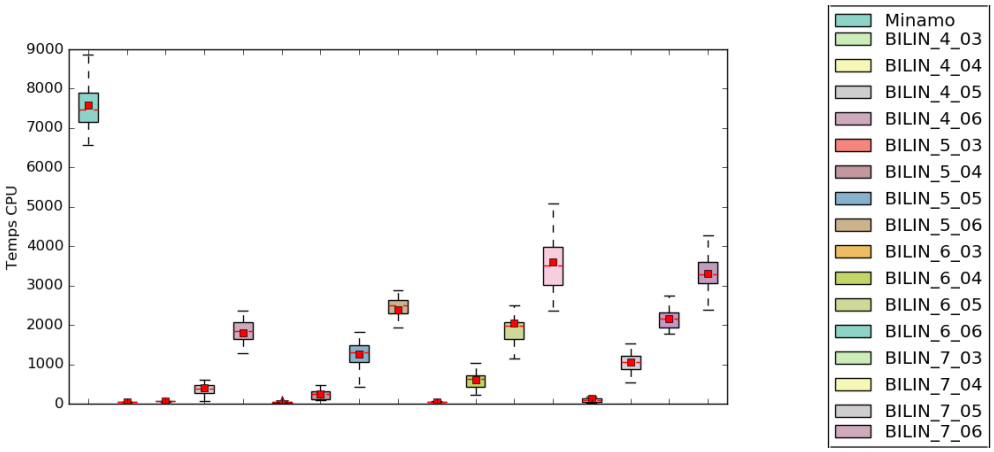


(b) *Rosenbrock 10D* - Temps d'exécution

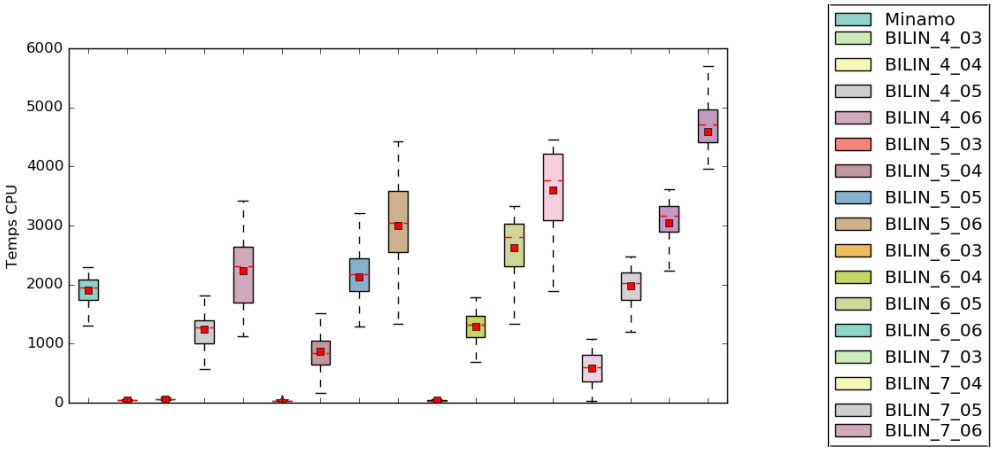


(c) *Rastrigin 10D* - Temps d'exécution

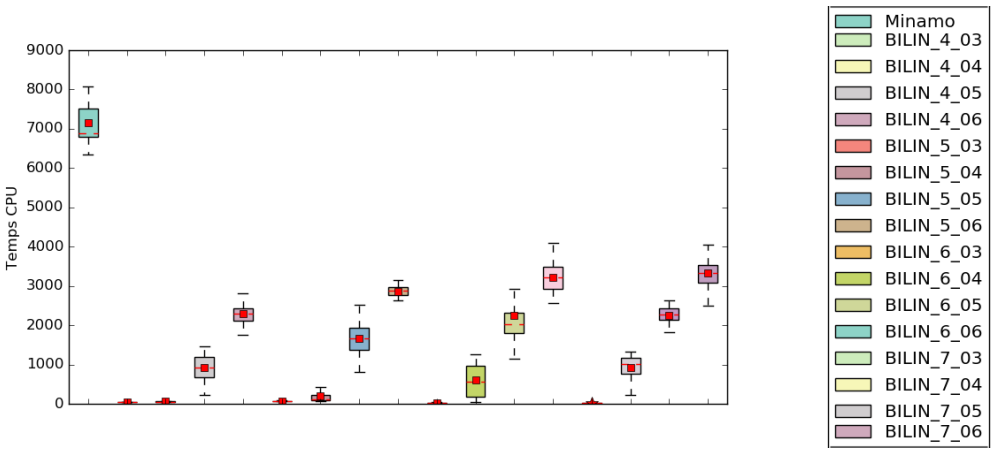
FIGURE 3 – Ces graphiques représentent le temps d'exécution en seconde des différentes stratégies *GAVaPS* contenant le calcul bi-linéaire, qui ont été exécutées avec l'AG pur sur les différents problèmes sans contraintes.



(a) *G7* - Temps d'exécution

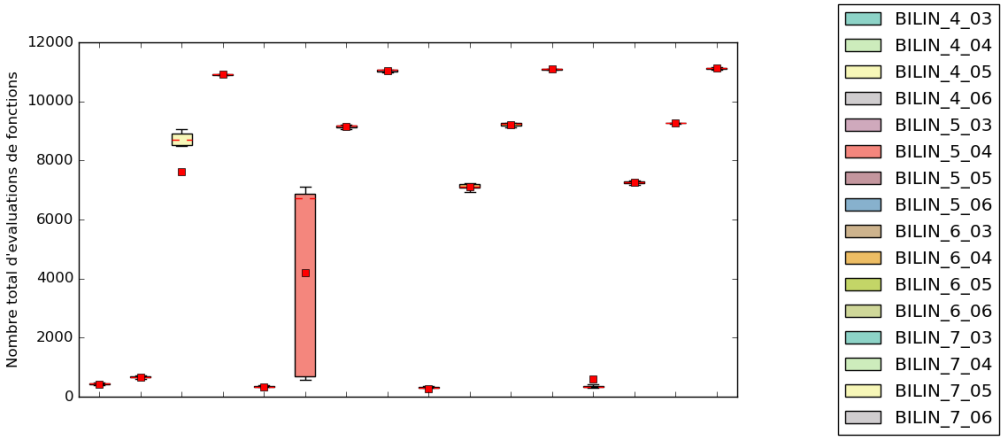


(b) *G10* - Temps d'exécution

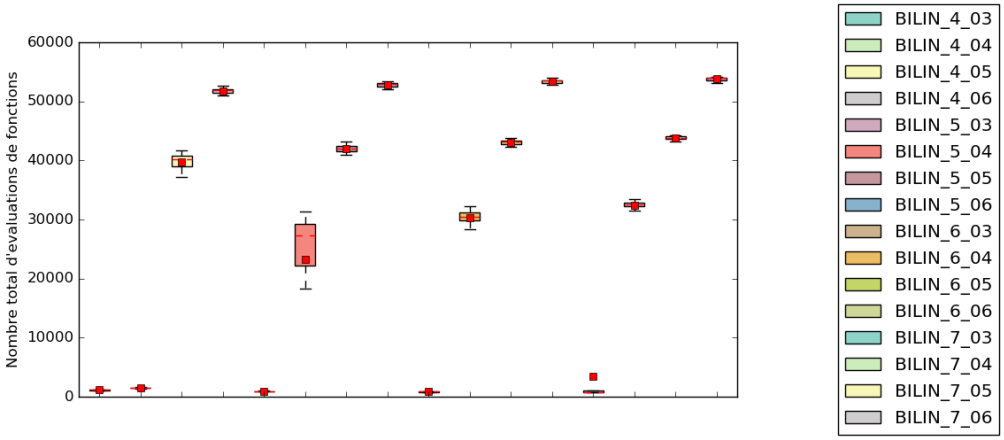


(c) *G2 plog 10* - Temps d'exécution

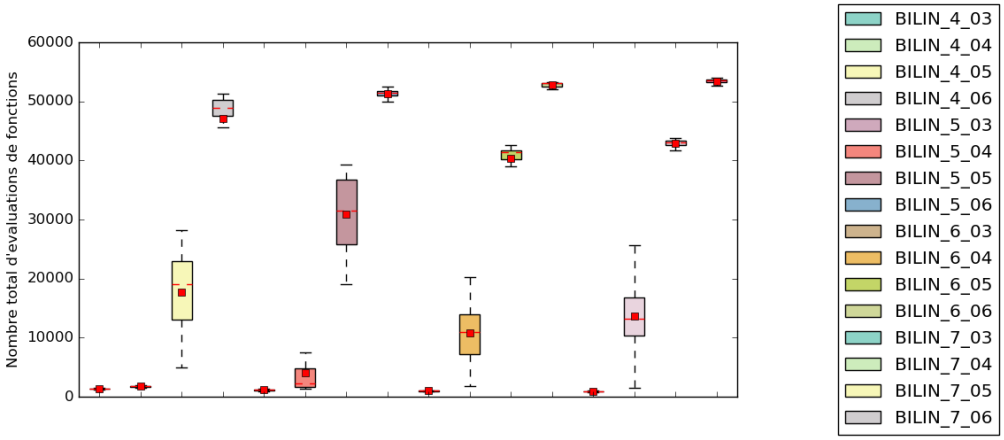
FIGURE 4 – Ces graphiques représentent le temps d'exécution en seconde des différentes stratégies *GAVaPS* contenant le calcul bi-linéaire, qui ont été exécutées avec l'AG pur sur les différents problèmes avec contraintes.



(a) *Rosenbrock 2D* - Nombre d'évaluations de fonctions (Minamo : 20'000 évaluations)

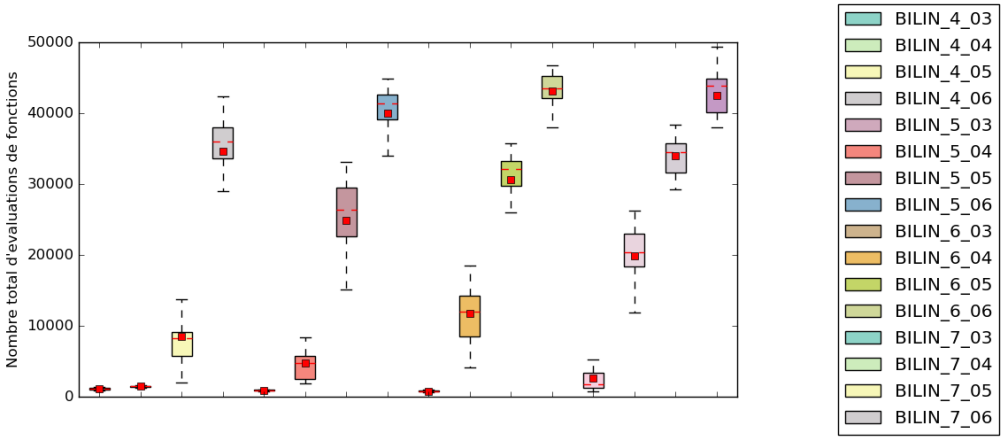


(b) *Rosenbrock 10D* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

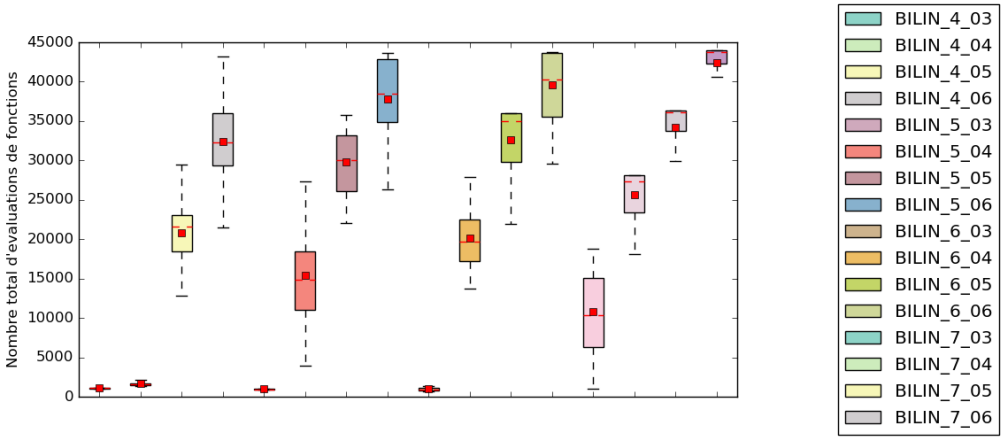


(c) *Rastrigin 10D* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

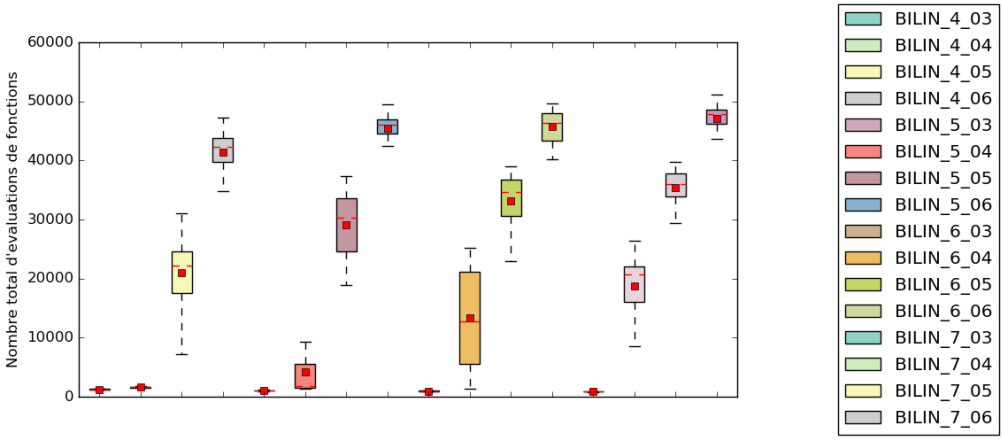
FIGURE 5 – Ces graphiques représentent le nombre d'évaluations de fonctions des différentes stratégies *GAVaPS* contenant le calcul bi-linéaire, qui ont été exécutées avec l'AG pur sur les différents problèmes sans contraintes. Le nombre d'évaluations de fonctions par Minamo de chaque méthode est repris dans la description de chaque problème.



(a) *G7* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

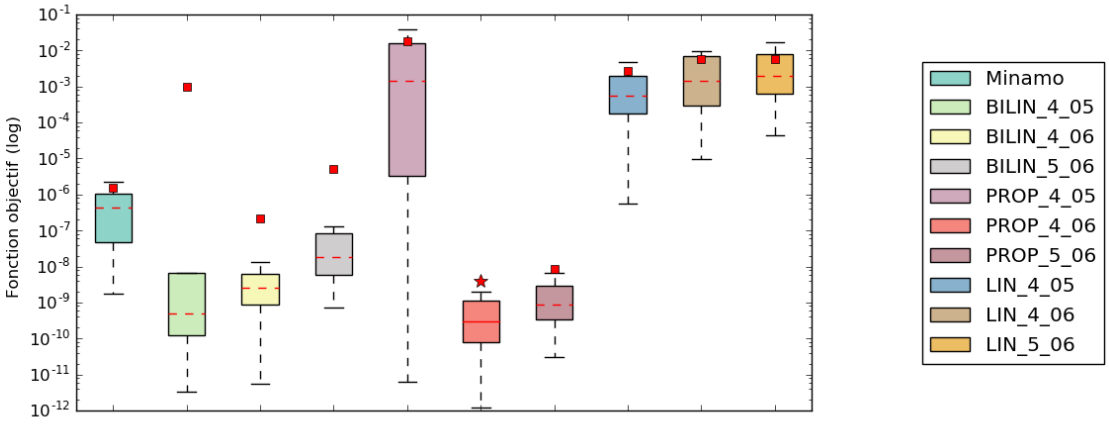


(b) *G10* - Nombre d'évaluations de fonctions (Minamo : 80'000 évaluations)

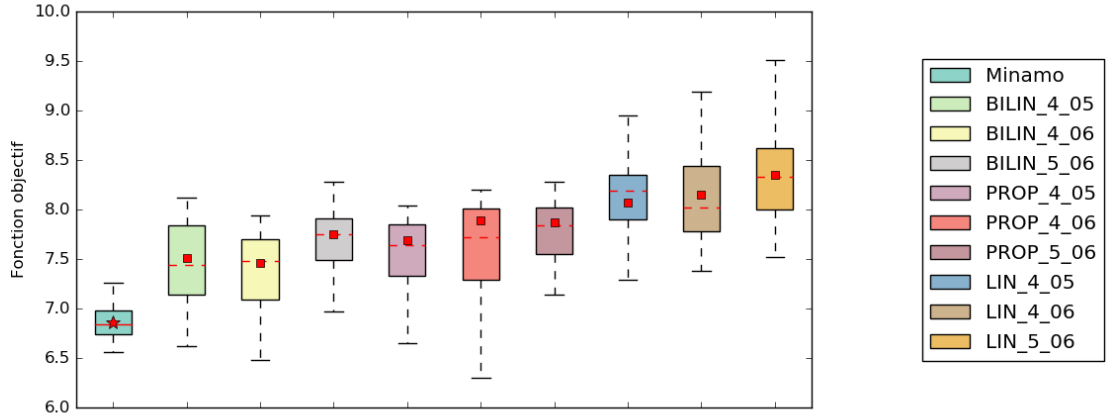


(c) *G2 plog 10* - Nombre d'évaluations de fonctions (Minamo : 100'000 évaluations)

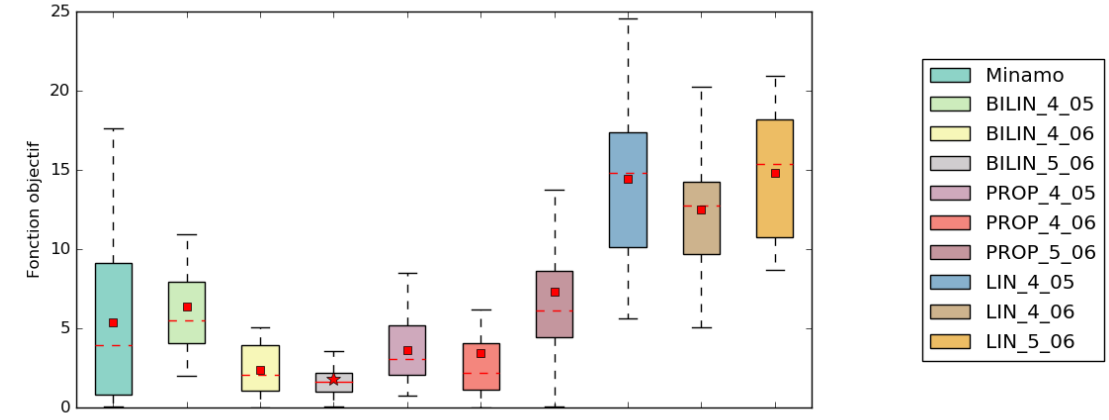
FIGURE 6 – Ces graphiques représentent le nombre d'évaluations de fonction sdes différentes stratégies *GAVaPS* contenant le calcul bi-linéaire, qui ont été exécutées avec l'AG pur sur les différents problèmes avec contraintes. Le nombre d'évaluations de fonctions par Minamo de chaque méthode est repris dans la description de chaque problème.



(a) *Rosenbrock 2D* - Valeurs finales de la fonction objectif

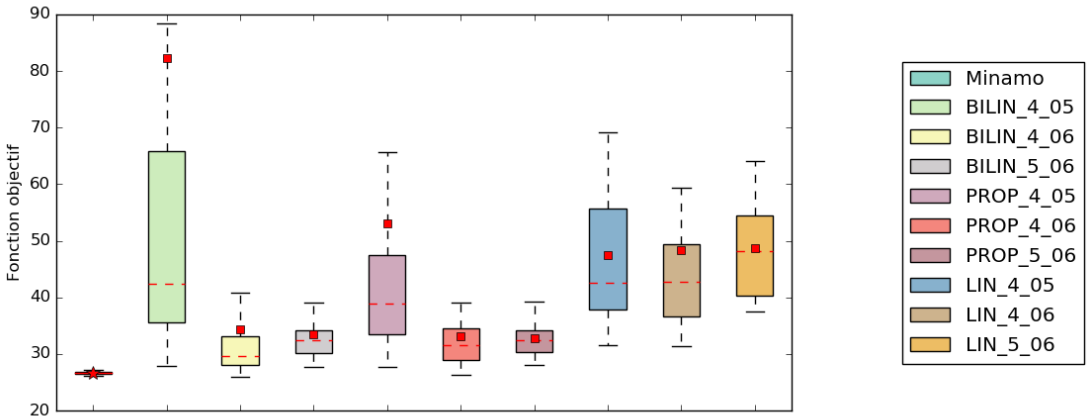


(b) *Rosenbrock 10D* - Valeurs finales de la fonction objectif

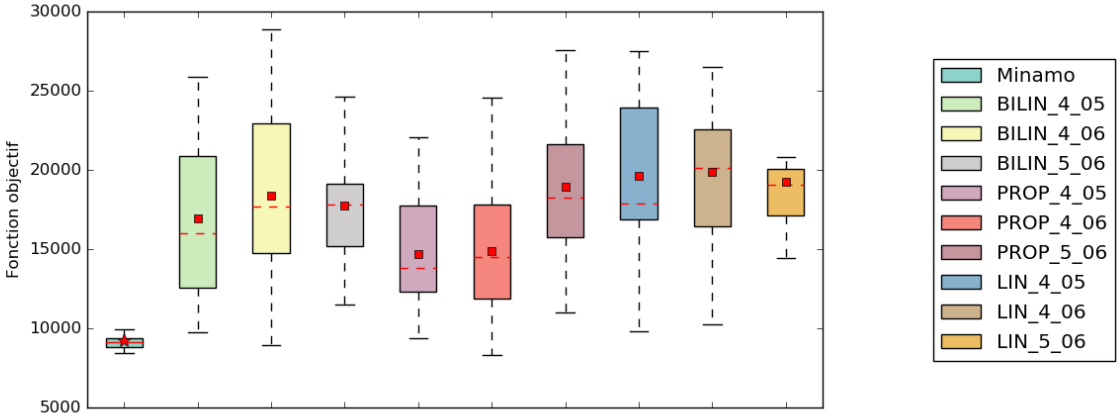


(c) *Rastrigin 10D* - Valeurs finales de la fonction objectif

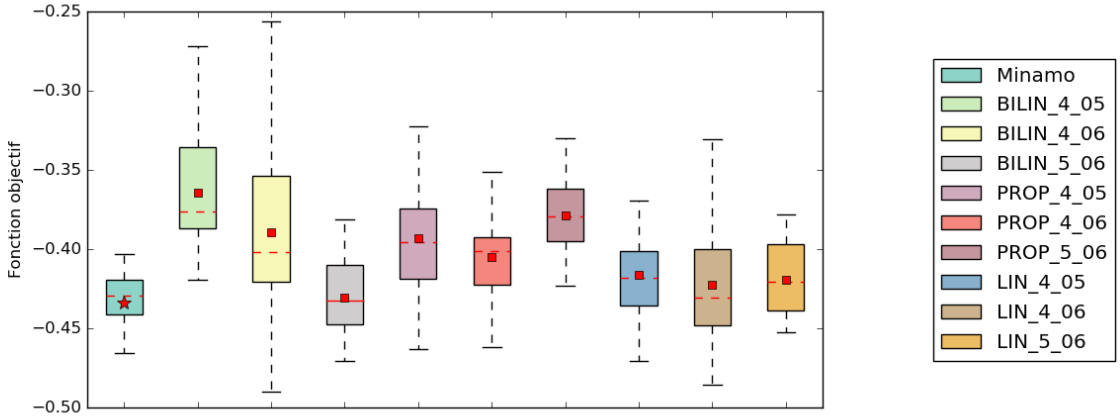
FIGURE 7 – Ces graphiques représentent la convergence finale des différentes combinaisons pour *GAVaPS* qui ont été exécutées avec l'AG pur sur les différents problèmes sans contraintes.



(a) *G7* - Valeurs finales de la fonction objectif

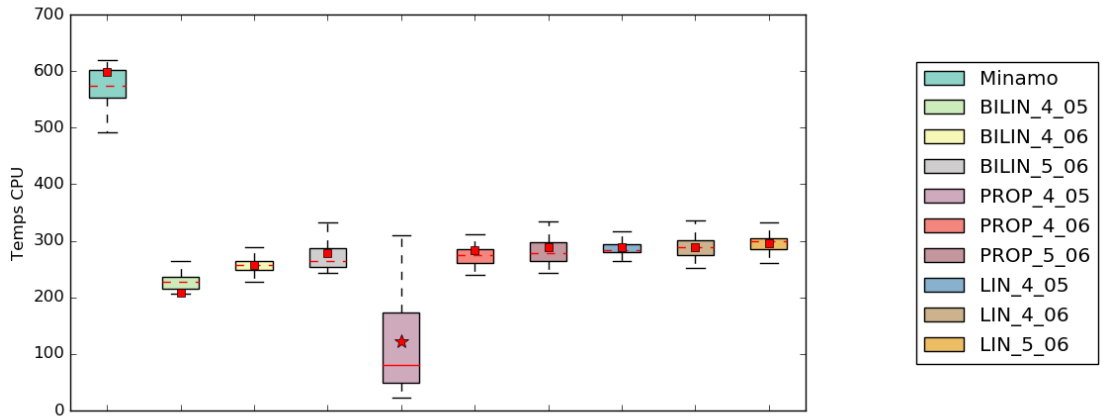


(b) *G10* - Valeurs finales de la fonction objectif

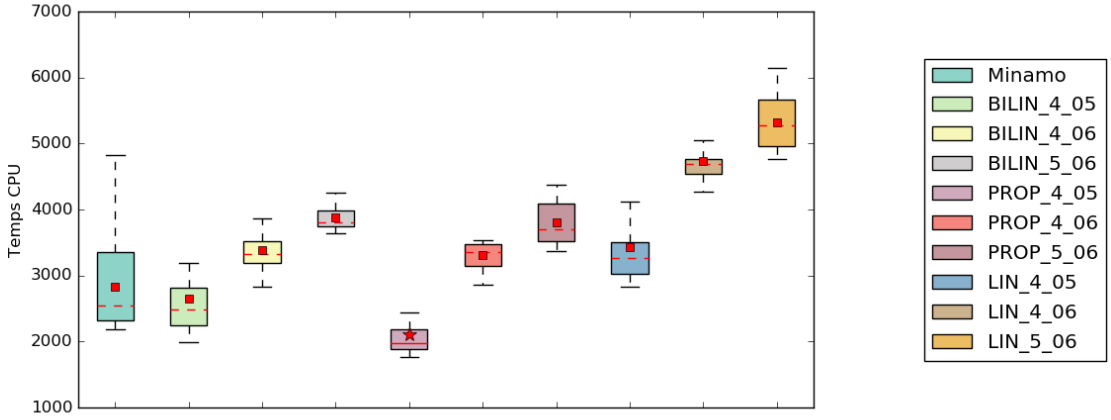


(c) *G2 plog 10* - Valeurs finales de la fonction objectif

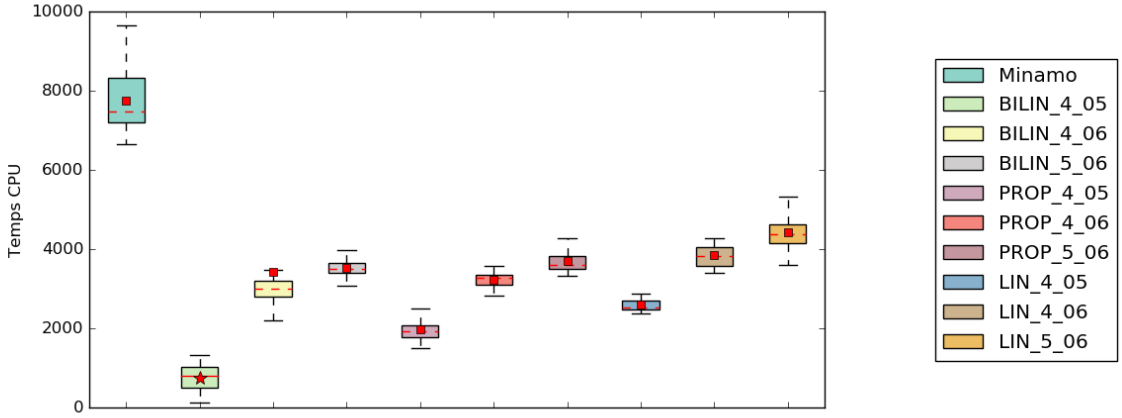
FIGURE 8 – Ces graphiques représentent la convergence finale des différentes combinaisons pour *GAVaPS* qui ont été exécutées avec l'AG pur sur les différents problèmes avec contraintes.



(a) *Rosenbrock 2D* - Temps d'exécution

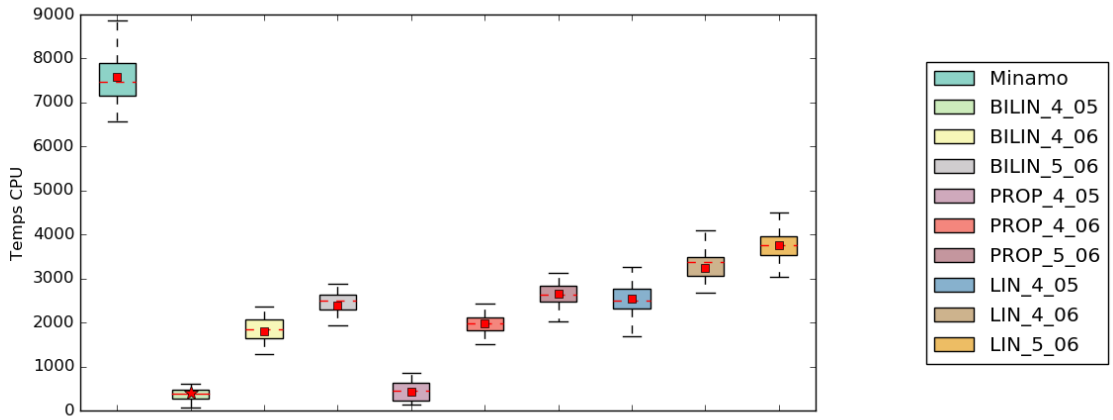


(b) *Rosenbrock 10D* - Temps d'exécution

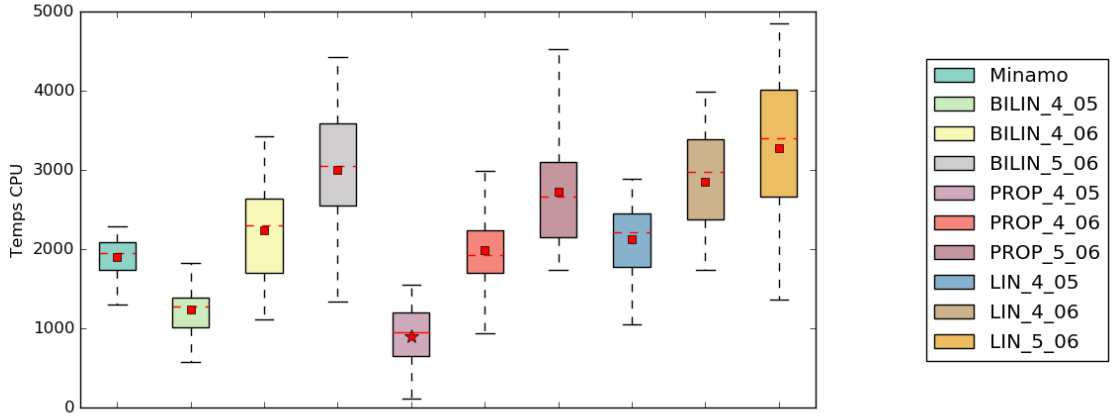


(c) *Rastrigin 10D* - Temps d'exécution

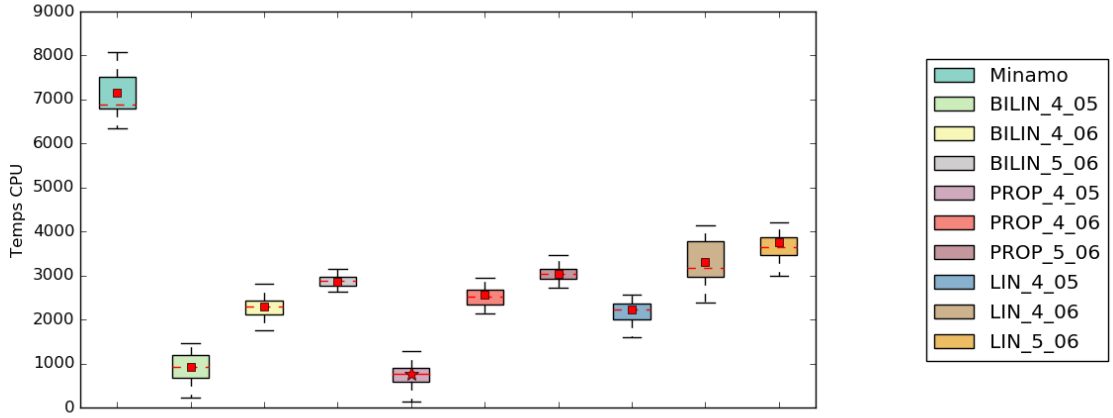
FIGURE 9 – Ces graphiques représentent le temps d'exécution en seconde des différentes combinaisons pour *GAVaPS* qui ont été exécutées avec l'AG pur sur les différents problèmes sans contraintes.



(a) *G7* - Temps d'exécution



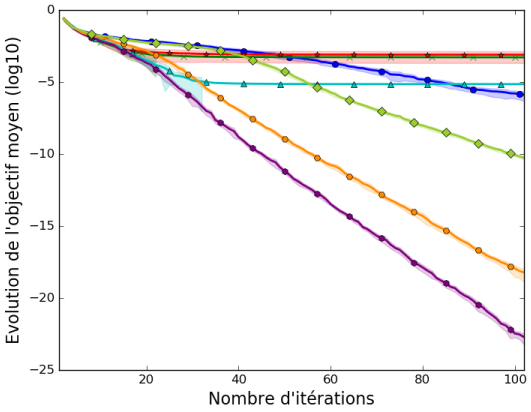
(b) *G10* - Temps d'exécution



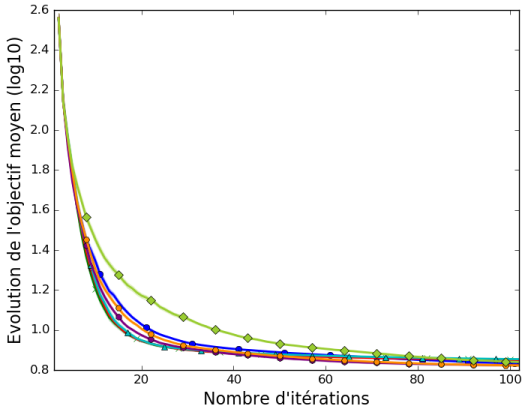
(c) *G2 plog 10* - Temps d'exécution

FIGURE 10 – Ces graphiques représentent le temps d'exécution en seconde des différentes combinaisons pour *GAVaPS* qui ont été exécutées avec l'AG pur sur les différents problèmes avec contraintes.

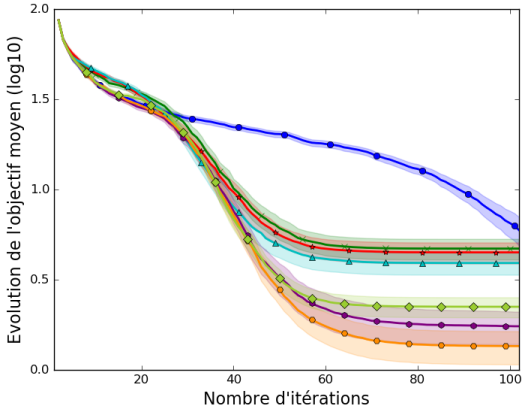
Annexe B. *FiScIS-EA*



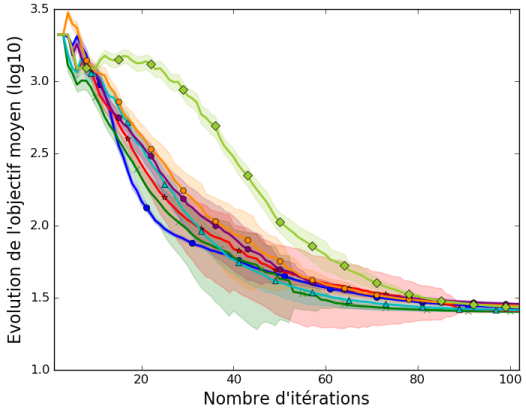
(a) *Rosenbrock 2D* - Graphe de convergence



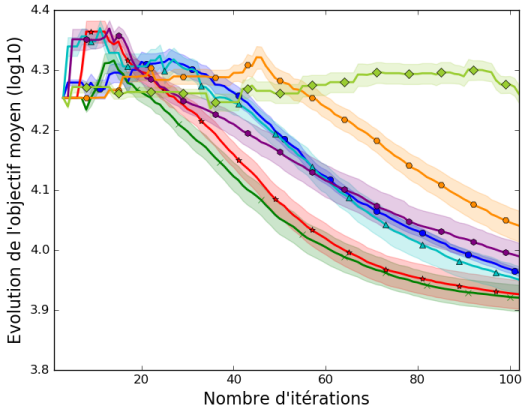
(b) *Rosenbrock 10D* - Graphe de convergence



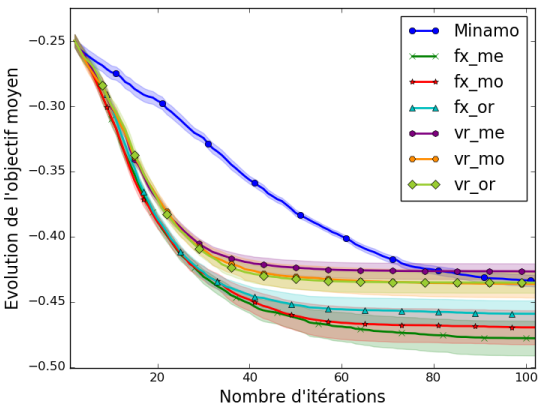
(c) *Rastrigin 10D* - Graphe de convergence



(d) *G7* - Graphe de convergence

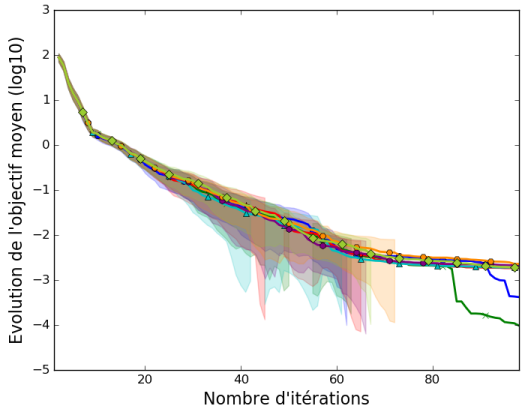


(e) *G10* - Graphe de convergence

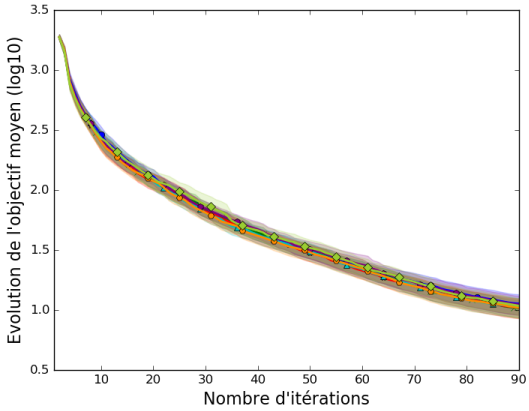


(f) *G2 plog 10* - Graphe de convergence

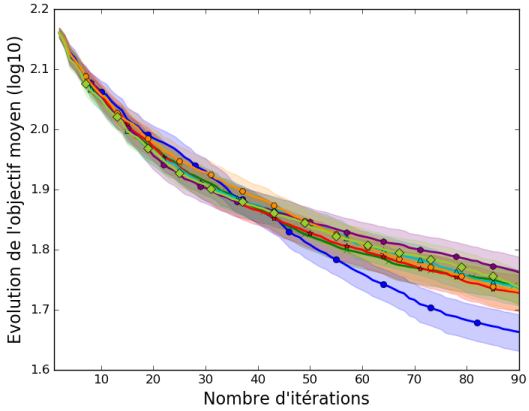
FIGURE 11 – Ces graphes représentent la convergence moyenne pour les différentes versions testées de *FiScIS-EA* et Minamo sur les six cas tests avec l'AG pur.



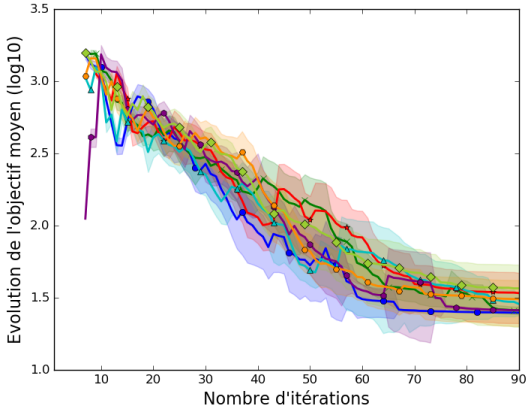
(a) *Rosenbrock 2D* - Graphe de convergence



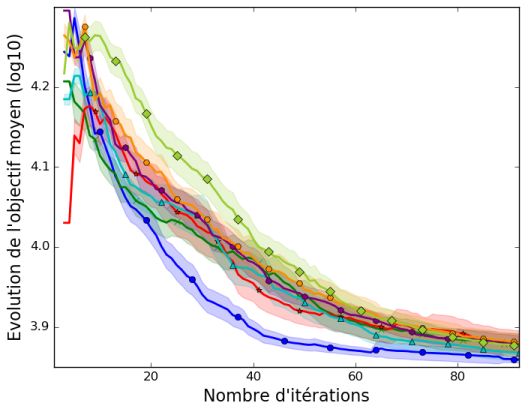
(b) *Rosenbrock 10D* - Graphe de convergence



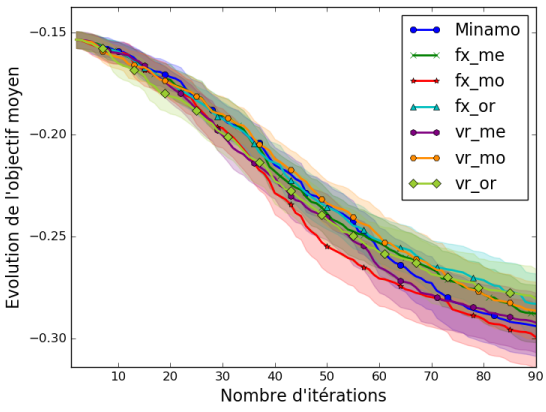
(c) *Rastrigin 10D* - Graphe de convergence



(d) *G7* - Graphe de convergence



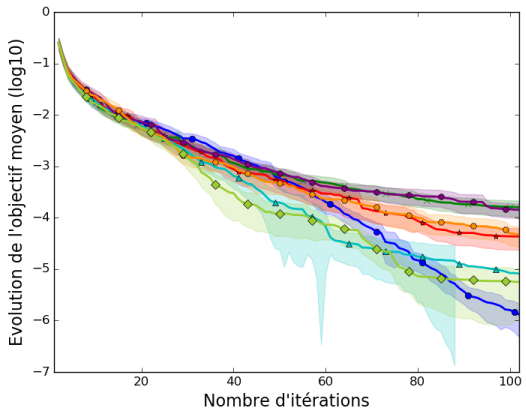
(e) *G10* - Graphe de convergence



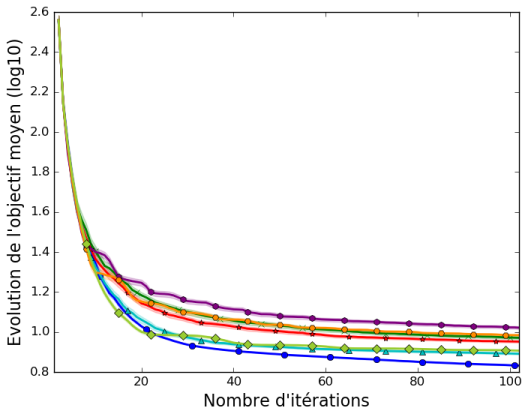
(f) *G2 plog 10* - Graphe de convergence

FIGURE 12 – Ces graphes représentent la convergence moyenne pour les différentes versions testées de *FiScIS-EA* et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*.

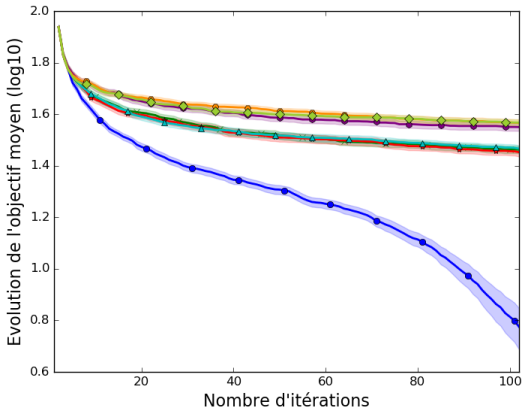
Annexe C. *FiScIS-EA* en dent de scie



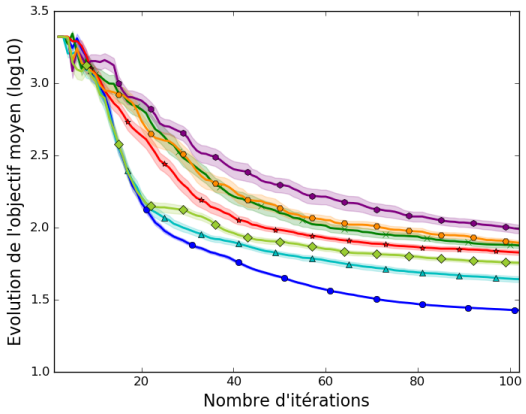
(a) *Rosenbrock 2D* - Graphe de convergence



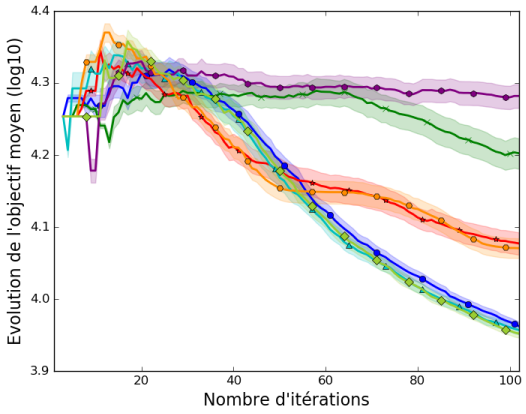
(b) *Rosenbrock 10D* - Graphe de convergence



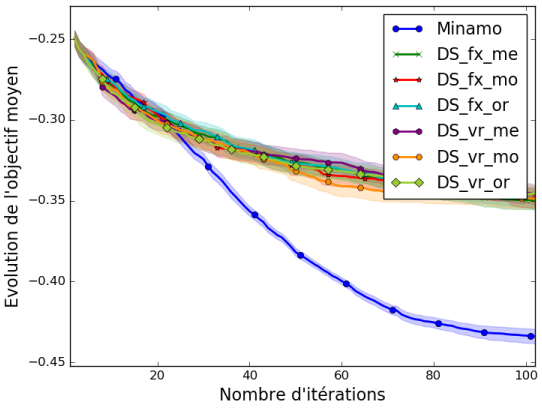
(c) *Rastrigin 10D* - Graphe de convergence



(d) *G7* - Graphe de convergence

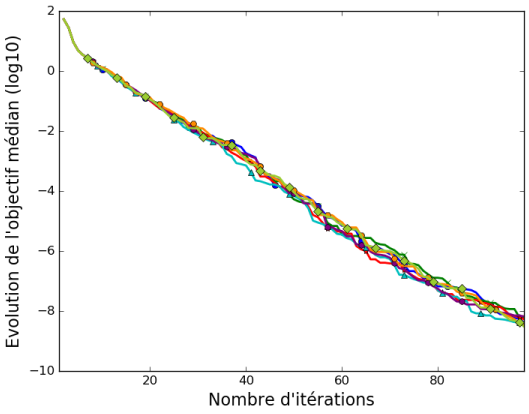


(e) *G10* - Graphe de convergence

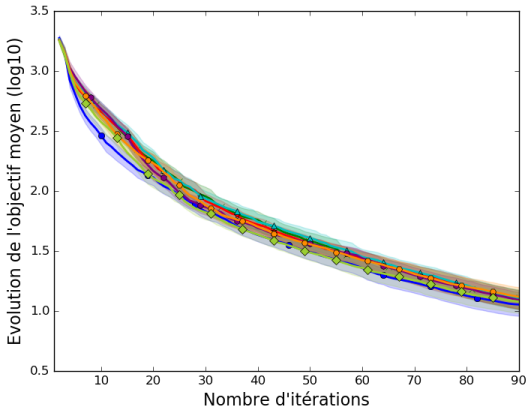


(f) *G2 plog 10* - Graphe de convergence

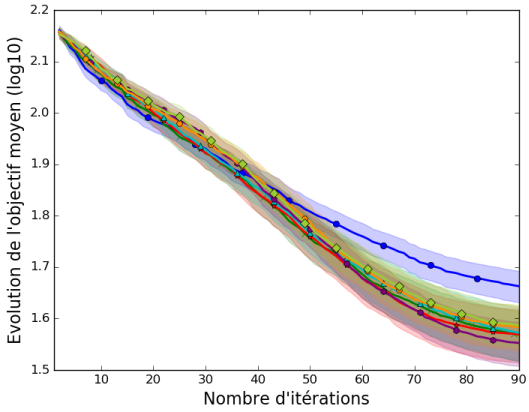
FIGURE 13 – Ces graphes représentent la convergence moyenne pour les différentes versions testées de *FiScIS-EA* en dent de scie et Minamo sur les six cas tests avec l'AG pur.



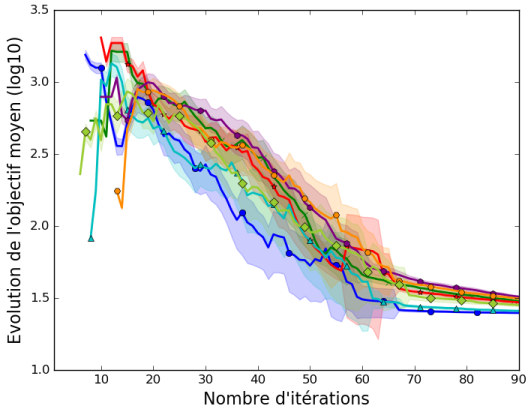
(a) *Rosenbrock 2D* - Graphe de convergence



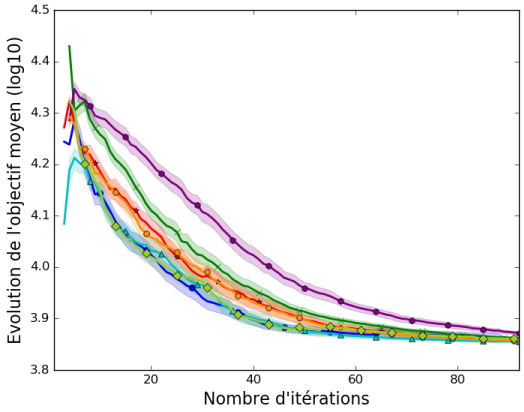
(b) *Rosenbrock 10D* - Graphe de convergence



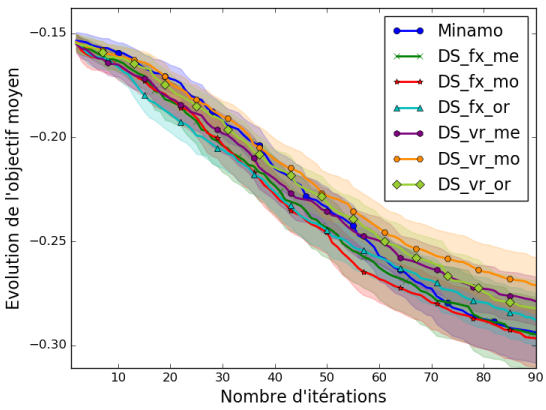
(c) *Rastrigin 10D* - Graphe de convergence



(d) *G7* - Graphe de convergence



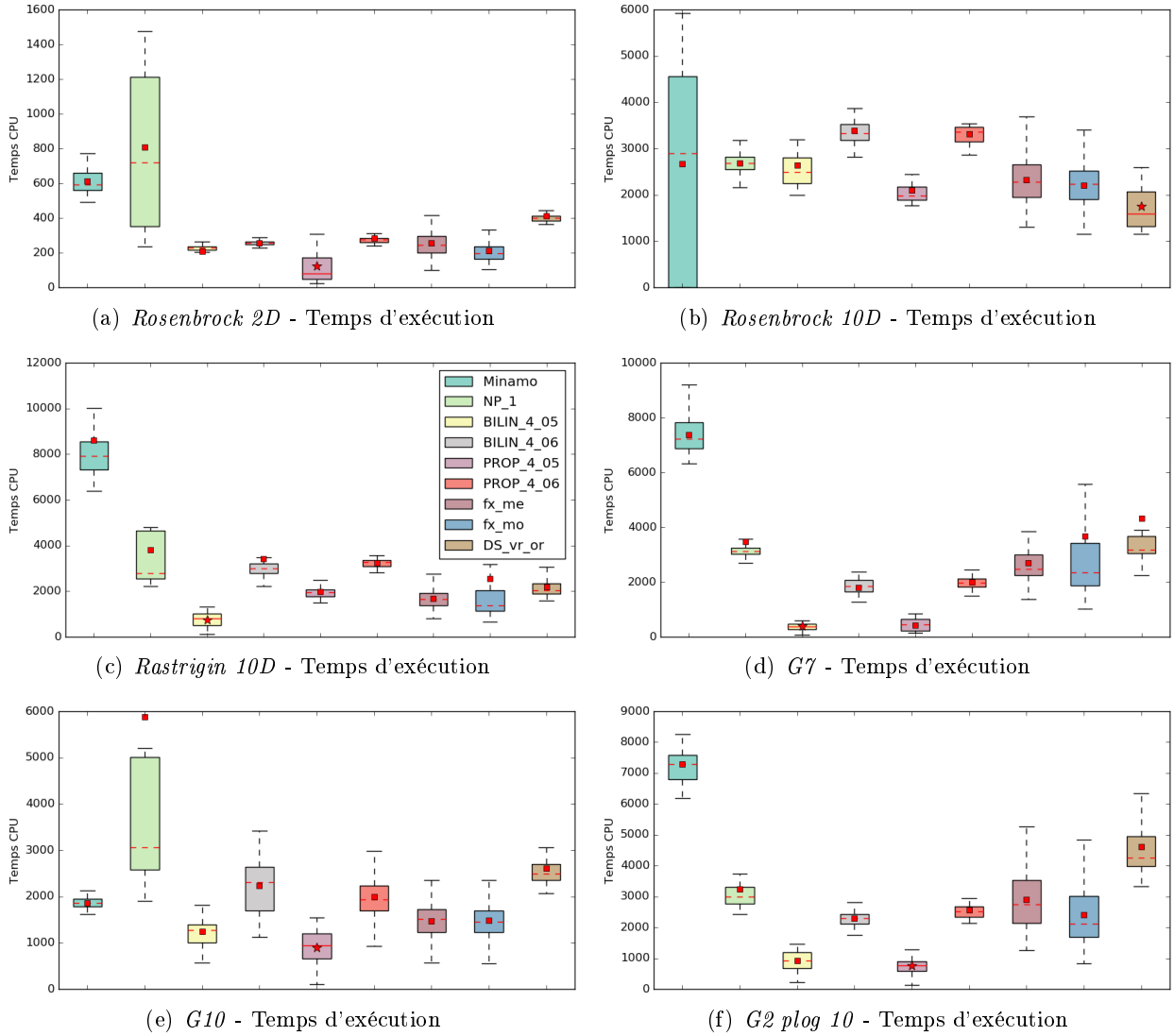
(e) *G10* - Graphe de convergence



(f) *G2 plog 10* - Graphe de convergence

FIGURE 14 – Ces graphes représentent la convergence moyenne pour les différentes versions testées de *FiScIS-EA* en dent de scie et Minamo sur les six cas tests avec l'AG intégré dans une boucle *SBO*.

Annexe D. Comparatif des méthodes



(a) *Rosenbrock 2D* - Temps d'exécution

(b) *Rosenbrock 10D* - Temps d'exécution

(c) *Rastrigin 10D* - Temps d'exécution

(d) *G7* - Temps d'exécution

(e) *G10* - Temps d'exécution

(f) *G2 plog 10* - Temps d'exécution

FIGURE 15 – Ces boîtes à moustache représentent les résultats synthétiques des différentes versions retenues au cours de ce mémoire et Minamo sur les six cas tests avec l'AG pur. Celles-ci concernent le temps d'exécution en seconde. La légende est la même pour toutes ces figures.

Liste des errata présents dans le mémoire

Youri DERLET

24 juin 2019

Dans ce document nous allons répertorier les errata présents dans ce mémoire. Nous allons commencer par clarifier la formule de *FiScIS-EA* présentée en page 66 de ce mémoire, qui n'est pas celle utilisée dans l'implémentation. Ensuite, nous citerons quelques errata mineurs.

1 *FiScIS-EA* - la formule originale

La formule originale pour le calcul de la probabilité de survie, présentée en page 66 de ce mémoire (Section 1 du Chapitre 6), n'est pas celle utilisée dans l'implémentation. En effet, celle-ci est un simple copier-coller de la présentation des réglages existants (Chapitre 2) qui concernent les problèmes à maximiser. Or, Minamo minimise des problèmes. Ainsi, la formule est traduite comme suit :

$$P_{surv}(i) = \begin{cases} \dfrac{fit_{max}(t) - fit(i)}{fit_{max}(t) - fit_{min}(t)} & \text{si } fit_{min}(t) \neq fit_{max}(t) \\ 1 & \text{sinon} \end{cases} \tag{1}$$

où $fit_{min}(t)$ et $fit_{max}(t)$ sont toujours les valeurs de fitness minimales et maximales observées dans la population $P(t)$, pour une itération t. Ainsi, l'explication algorithmique présentée en page 66 est correcte. Pour rappel, celle-ci était : pour un nombre aléatoire q tiré entre zéro et un, si celui-ci est plus petit que $P_{surv}(i)$ alors l'individu i survit, sinon il est supprimé de la population. En effet, plus l'individu est bon, plus sa valeur de fitness est proche de $fit_{min}(t)$, ainsi sa probabilité de survie est proche de 1.

Nous tenons à préciser que
- la bonne formule (Formule 1) a été implémentée dans Minamo ;
- tous les résultats issus des versions contenant le tag **or** présentées dans ce mémoire proviennent bien de la Formule 1 ;
- la Formule 6.1 (page 66 du mémoire) est une erreur, car elle n'est pas applicable pour la minimisation de fonctions.

Nous nous excusons pour cette erreur.

2 Quelques errata mineurs

Nous avons également découverts quelques errata. Ceux-ci sont mineurs par rapport à la formule originale de *FiScIS-EA*. Premièrement, dans le Glossaire en page 3 et 4 de ce mémoire, la variable $\overline{T}$ est définie deux fois. En réalité, la première fois qu'elle est définie cette variable est $\bar{t}$. La seconde fois, cette variable est correcte : elle est bien $\overline{T}$. Ensuite, dans le dernier paragraphe de la Section 5.3, il y a deux fois PROP_4_05 dans les quatre méthodes retenues. Or, les quatre versions sont bien BILIN_4_05, BILIN_4_06, PROP_4_05 et PROP_4_06, comme l'explication le suggère. Pour finir, dans l'avant-dernier paragraphe de la Conclusion, nous avons oublié de préciser que la version de *Saw-Tooth* contenant trois périodes est la méthode retenue pour le *SBO* en non pour l'AG pur. Nous nous excusons pour ces errata et autres coquilles que nous n'avons pas remarquées.

Analyses statistiques et vérification de nos résultats présents dans le mémoire sur cinq autres problèmes de minimisation

Youri DERLET

24 juin 2019

Dans ce document, nous allons expliquer les ajouts que nous avons réalisés pour la défense orale de ce mémoire. Nous commencerons par citer les cinq cas tests supplémentaires, dont deux concernent des cas physiques. Ensuite, nous expliquerons et analyserons les statistiques utilisées pour vérifier quelles sont les méthodes qui améliorent Minamo, d'un point de vue scientifique (en considérant les solutions en fin d'exécution) et d'un point de vue applicatif (en considérant le temps d'exécution CPU en seconde). Pour finir, nous présenterons le graphe de performance (avec un seuil fin) pour les onze cas tests résolus avec les meilleures méthodes retenues dans le Chapitre 7 du mémoire.

1 Cinq cas tests supplémentaires

Afin de vérifier que nous n'avions pas promu (dans le Chapitre 7 du mémoire) des méthodes de résolution pour un sous-ensemble de problèmes (présentés dans la Section 4 du Chapitre 3), nous avons choisi de vérifier nos résultats sur cinq autres cas tests, dont deux représentent des problèmes physiques. Ceux-ci sont repris dans le Tableau 1 avec le nombre de dimension n_D de chaque problème et deux propriétés : premièrement, si le problème est contraint ou non ; deuxièmement, si le problème a un caractère industriel (physique) ou non.

Problème	n_D	Contraint	Industriel
Pressure Vessel	4	oui	oui
Speed Reducer	7	oui	oui
Ackley	10	non	non
Branin Custom	2	oui	non
Schwefel	10	non	non

TABLE 1 – Tableau reprenant les propriétés des cinq cas tests supplémentaires.

Le problème industriel *Pressure Vessel* est un récipient conçu pour contenir un gaz ou un liquide dont la pression interne est différente de la pression atmosphérique externe. Les paramètres de ce problème sont notamment la température et la pression. Une modélisation scientifique de ce problème est connu et est :

$$f(x_1, \cdots, x_4) = 0.6224 x_1 x_3 x_4 + 1.7781 x_2 x_3^2 + 3.1661 x_1^2 x_4 + 19.84 x_1^2 x_3,$$

pour tout $x_1, x_2 \in [0, 1],\ x_3 \in [0, 50],\ x_4 \in [0, 240]$. Ses contraintes sont :

$$\begin{aligned} c_1(x_1, \cdots, x_4) &\equiv -x_1 + 0.0193 x_3 \leq 0 \\ c_2(x_1, \cdots, x_4) &\equiv -x_2 + 0.00954 x_3 \leq 0 \\ c_3(x_1, \cdots, x_4) &\equiv -\pi x_3^2 x_4 - \tfrac{4}{3}\pi x_3^3 + 1296000 \leq 0 \end{aligned}$$

La meilleure solution connue est $f(x_1, \cdots, x_4) = 5804.45$. La solution cible est $f(x_1, \cdots, x_4) = 6000.0$.

Le second problème industriel, *Speed Reducer*, est un mécanisme de réduction de la vitesse qui permet de décaler la vitesse d'un rotor par exemple. Les applications les plus concrètes sont les moteurs de voiture ou ceux des avions légers de tourisme. La traduction mathématique d'un tel problème est :

$$f(x_1, \cdots, x_7) = 0.7854 x_1 x_2^2 A - 1.508 x_1 B + 7.477 C + 0.7854 D,$$

où A, B, C et D sont des variables fixées à

$$\begin{array}{rcl} A & = & 3.3333 x_3^2 + 14.9334 x_3 - 43.0934 \\ B & = & x_6^2 + x_7^2 \\ C & = & x_6^3 + x_7^3 \\ D & = & x_4 x_6^2 + x_5 x_7^2 \end{array}$$

et défini pour tout $x_1 \in [2.6, 3.6]$, $x_2 \in [0.7, 0.8]$, $x_3 \in [17, 28]$, x_4 et $x_5 \in [7.3, 8.3]$, $x_6 \in [2.6, 3.9]$, $x_7 \in [5, 5.5]$. Ses contraintes sont :

$$\begin{array}{rcl} c_1(x_1, \cdots, x_7) & \equiv & 27 - x_1 x_2 x_3 \leq 0 \\ c_2(x_1, \cdots, x_7) & \equiv & 397.5 - x_1 x_2^2 x_3^2 \leq 0 \\ c_3(x_1, \cdots, x_7) & \equiv & 1.93 - \frac{x_2 x_3 x_6^4}{x_4^3} \leq 0 \\ c_4(x_1, \cdots, x_7) & \equiv & 1.93 - \frac{x_2 x_3 x_7^4}{x_5^3} \leq 0 \\ c_5(x_1, \cdots, x_7) & \equiv & \frac{A_1}{B_1} - 1100 \leq 0 \\ c_6(x_1, \cdots, x_7) & \equiv & \frac{A_2}{B_2} - 850 \leq 0 \\ c_7(x_1, \cdots, x_7) & \equiv & x_2 x_3 - 40 \leq 0 \\ c_8(x_1, \cdots, x_7) & \equiv & 5 - \frac{x_1}{x_2} \leq 0 \\ c_9(x_1, \cdots, x_7) & \equiv & \frac{x_1}{x_2} - 12 \leq 0 \\ c_{10}(x_1, \cdots, x_7) & \equiv & 1.9 + 1.5 x_6 - x_4 \leq 0 \\ c_{11}(x_1, \cdots, x_7) & \equiv & 1.9 + 1.1 x_7 - x_5 \leq 0 \end{array}$$

où

$$\begin{array}{rcl} A_1 & = & \sqrt{\frac{745 x_4}{x_2 x_3}^2 + (16.91 \times 10^6)} \\ B_1 & = & 0.1 x_6^3 \\ A_2 & = & \sqrt{\frac{745 x_5}{x_2 x_3}^2 + (157.5 \times 10^6)} \\ B_2 & = & 0.1 x_7^3 \end{array}$$

La meilleure solution est $f(x_1, \cdots, x_7) = 2994.42$. Tandis que la valeur cible est $f(x_1, \cdots, x_7) = 2995.0$.

Les définitions des trois autres problèmes sont des problèmes communs. Ainsi, nous ne les décrirons pas dans ces pages.

2 Tests statistiques

Dans cette section, nous expliquerons d'abord les tests statistiques que nous avons réalisés en plus de ce mémoire pour notre défense orale. Ensuite, nous reprendrons les résultats statistiques. Ceux-ci permettent de montrer avec certitude si une version d'une méthode améliore ou détériore Minamo plus qu'une autre version.

2.1 Explication

Afin de tester si une méthode améliore ou détériore Minamo, nous avons réalisé trois tests statistiques consécutifs. En effet, le but de cette démarche est de s'assurer statistiquement que les méthodes permettent l'amélioration de la solution finale de Minamo et du temps d'exécution. Pour ce faire, nous disposons de cent exécutions différentes de chaque méthode sur six ou onze cas tests. Comme le cours d'Analyse Multivarié donné à l'Université de Namur en Mathématique le suggère, trois tests scientifiques consécutifs sont nécessaire pour mettre en évidence une différence significative entre les moyennes d'un ensemble par rapport à un autre. Ceux-ci sont :

1. le test de variance de Bartlett ;
2. si le test précédent démontre que les variances ne sont pas significativement différentes, pour un niveau du test de 0.05, alors le test ANOVA est appliqué, sinon un test non-paramétrique de Kruskal et Wallis est appliqué ;
3. si un test précédent démontre qu'au moins une méthode induit une moyenne significativement différente à une autre, pour un niveau de test de 0.05, alors les contrastes de Scheffé sont réalisés.

Le test de variance de Bartlett détermine si les résultats des différentes méthodes testées (et Minamo compris) possèdent la même variance. Le second test détermine si les moyennes des résultats de chaque méthode sont semblables ou non. Pour finir, sur un cas test donné, les contrastes de Scheffé permettent de déterminer les méthodes qui améliorent significativement Minamo, détériorent significativement Minamo, ou ne sont pas significativement différentes (statu quo).

2.2 Résultats

Comme un objectif de ce mémoire a été de démontrer que la méthode originale de *FiScIS-EA* introduit par Cook et Tauritz [1] pouvait être améliorée, nous allons commencer par montrer statistiquement cette amélioration (ou non détérioration) en terme de solutions finales et en terme de temps d'exécution. Ensuite, nous montrerons que ces adaptations améliorent le plus la valeur finale parmi les versions testées dans ce mémoire et retenues dans le Chapitre 7.

Dans les Tableaux 2 et 3, les résultats synthétiques des contrastes de Scheffé pour les six cas tests résolus avec l'AG pur sont représentés et permettent de comparer les versions de *FiScIS-EA* de ce mémoire. De plus, la comparaison est faite avec Minamo. Notons que les deux premiers tests statistiques nous autorisaient à réaliser ces contrastes. Dans le premier tableau, la comparaison est faite sur la solution finale moyenne des cent exécutions de chaque méthode. Par contre, dans le second tableau, le temps d'exécution est la caractéristique observée. Par exemple, la version originale de *FiScIS-EA* (*vr or*) améliore statistiquement la solution de deux cas tests, la détériore pour un cas test et la solution est identique pour la moitié des cas tests (Tableau 2). Nous constatons donc que les méthodes *fx mo* et *fx me* permettent d'améliorer Minamo par rapport à la solution (ou ne pas la détériorer) pour cinq cas tests sur les six (Tableau 2) et que le temps d'exécution est toujours amélioré par rapport à Minamo, en comparaison avec la méthode originale qui la détériore même pour la moitié des exécutions (Tableau 3).

Dans les Tableaux 4 et 5, les résultats synthétiques concernent toujours les versions de *FiScIS-EA*. Cependant, ceux-ci concernent la résolution avec le *SBO*. Nous constatons à nouveaux, que les deux versions améliorées *fx mo* et *fx me* permettent une non-détérioration de la solution par rapport à Minamo (Tableau 4) et un gain en temps ou une non-détérioration (Tableau 5). Notons que la version originale de *FiScIS-EA* ne permet pas d'améliorer notablement Minamo, aussi bien pour la solution que pour le temps d'exécution par rapport aux deux versions retenues.

Dans les Tableaux 6 et 7, les meilleures méthodes du Chapitre 7 de ce mémoire sont comparées avec les onze cas tests en utilisant les contrastes de Scheffé. Ces résultats concernent l'AG pur. Nous

constatons que les versions améliorées de *FiScIS-EA* (*fx me* et *fx mo*) améliorent (ou ne détériorent pas) considérable la solution finale par rapport à Minamo (Tableau 6) et améliorent également le temps d'exécution pour tous les cas tests (Tableau 7). Ainsi, ces deux versions représentent les meilleures méthodes de ce mémoire pour l'AG pur.

Dans les Tableaux 8 et 9, ce sont les versions exécutées au sein de l'optimisation assistée par modèles de substitution. D'un point de vue scientifique, les versions *fx me* et *fx mo* améliorent (ou ne détériorent pas) Minamo par rapport aux autres versions testées en regardant la solution finale (Tableau 8), sans trop détériorer le temps d'exécution. Par contre, le temps d'exécution est toujours amélioré par rapport à Minamo pour les versions de *FiScIS-EA* en dent de scie retenues (*DS vr or* et *DS fx or*) et pour la version de *Saw-Tooth* à trois périodes $\overline{T} = 3$ (voir le Tableau 9) sans trop détériorer la solution finale. Ainsi, choisir une méthode parmi celles-ci peut être plus difficile.

	fx or	*fx mo*	*fx me*	*vr or*	*vr mo*	*vr me*
Amélioration(s) [A]	4	3	3	2	2	2
Détérioration(s)	1	1	1	1	0	0
Statu quo [S]	1	2	2	3	4	4
[A]+[S] sur 6	5/6	5/6	5/6	5/6	6/6	6/6

TABLE 2 – Ce tableau concerne l'AG pur et les versions de *FiScIS-EA*. Il reprend le nombre de cas tests qui ont été résolus avec une solution finale moyenne significativement améliorée par rapport à Minamo, détériorée ou non (statu quo). La version originale de *FiScIS-EA* est mise en évidence par une colonne grisée.

	fx or	*fx mo*	*fx me*	*vr or*	*vr mo*	*vr me*
Amélioration(s) [A]	4	6	6	3	3	5
Détérioration(s)	0	0	0	3	2	0
Statu quo [S]	2	0	0	0	1	1
[A]+[S] sur 6	6/6	6/6	6/6	3/6	4/6	6/6

TABLE 3 – Ce tableau concerne l'AG pur et les versions de *FiScIS-EA*. Il reprend le nombre de cas tests qui ont été résolus en un temps d'exécution moyen significativement amélioré par rapport à Minamo, détérioré ou non (statu quo). La version originale de *FiScIS-EA* est mise en évidence par une colonne grisée.

	fx or	*fx mo*	*fx me*	*vr or*	*vr mo*	*vr me*
Amélioration(s) [A]	1	1	1	1	0	1
Détérioration(s)	0	0	0	1	1	2
Statu quo [S]	5	5	5	4	5	3
[A]+[S] sur 6	6/6	6/6	6/6	5/6	5/6	5/6

TABLE 4 – Ce tableau concerne le *SBO* et les versions de *FiScIS-EA*. Il reprend le nombre de cas tests qui ont été résolus avec une solution finale moyenne significativement améliorée par rapport à Minamo, détériorée ou non (statu quo). La version originale de *FiScIS-EA* est mise en évidence par une colonne grisée.

	fx or	*fx mo*	*fx me*	*vr or*	*vr mo*	*vr me*
Amélioration(s) [A]	2	2	2	0	1	2
Détérioration(s)	3	2	2	6	5	4
Statu quo [S]	1	2	2	0	0	0
[A]+[S] sur 6	3/6	4/6	4/6	0/6	1/6	2/6

TABLE 5 – Ce tableau concerne le *SBO* et les versions de *FiScIS-EA*. Il reprend le nombre de cas tests qui ont été résolus en un temps d'exécution moyen significativement amélioré par rapport à Minamo, détérioré ou non (statu quo). La version originale de *FiScIS-EA* est mise en évidence par une colonne grisée.

	$\overline{T} = 1$	*fx me*	*fx mo*	*DS vr or*
Amélioration(s) [A]	2	8	7	1
Détérioration(s)	4	1	1	7
Statu quo [S]	5	2	3	3
[A]+[S] sur 11	7/11	10/11	10/11	4/11

TABLE 6 – Ce tableau concerne l'AG pur et toutes les versions retenues. Il reprend le nombre de cas tests qui ont été résolus avec une solution finale moyenne significativement améliorée par rapport à Minamo, détériorée ou non (statu quo).

	$\overline{T} = 1$	*fx me*	*fx mo*	*DS vr or*
Amélioration(s) [A]	6	10	11	10
Détérioration(s)	1	0	0	1
Statu quo [S]	4	1	0	0
[A]+[S] sur 11	10/11	11/11	11/11	10/11

TABLE 7 – Ce tableau concerne l'AG pur et toutes les versions retenues. Il reprend le nombre de cas tests qui ont été résolus en un temps d'exécution moyen significativement amélioré par rapport à Minamo, détérioré ou non (statu quo).

	$\overline{T} = 3$	*fx me*	*fx mo*	*DS vr or*	*DS fx or*
Amélioration(s) [A]	1	2	2	0	1
Détérioration(s)	3	0	0	2	2
Statu quo [S]	7	9	9	9	8
[A]+[S] sur 11	8/11	11/11	11/11	9/11	9/11

TABLE 8 – Ce tableau concerne le *SBO* et toutes les versions retenues. Il reprend le nombre de cas tests qui ont été résolus avec une solution finale moyenne significativement améliorée par rapport à Minamo, détériorée ou non (statu quo).

	$\overline{T} = 3$	*fx me*	*fx mo*	*DS vr or*	*DS fx or*
Amélioration(s) [A]	11	4	5	11	8
Détérioration(s)	0	4	2	0	0
Statu quo [S]	0	3	4	0	3
[A]+[S] sur 11	11/11	7/11	9/11	11/11	11/11

TABLE 9 – Ce tableau concerne le *SBO* et toutes les versions retenues. Il reprend le nombre de cas tests qui ont été résolus en un temps d'exécution moyen significativement amélioré par rapport à Minamo, détérioré ou non (statu quo).

3 Graphe de performance pour les onze cas tests

Dans cette section, le graphe de performance (avec un seuil fin) des méthodes testées en *SBO* sur onze cas tests (les six du mémoire et les cinq supplémentaires présentés dans cet ajout) est montré dans la Figure 1. Pour rappel, ce graphe de performance est une méthode synthétique développée par Dolan et More [2]. Il permet de synthétiser la performance de chaque méthode l'une par rapport à l'autre. En effet, une courbe située au dessus d'une autre est dite plus performante que cette seconde. Dans notre cas, il est clair que les méthodes améliorées de *FiScIS-EA* (*fx me* et *fx mo*) sont plus performantes que les autres stratégies et même que Minamo : elles sont plus robustes que Minamo ($\alpha = 5.0$) mais elles sont efficaces de la même manière que Minamo ($\alpha = 1.0$). Cependant, cette analyse ne concerne que l'axe scientifique, c'est à dire les solutions fines trouvées.

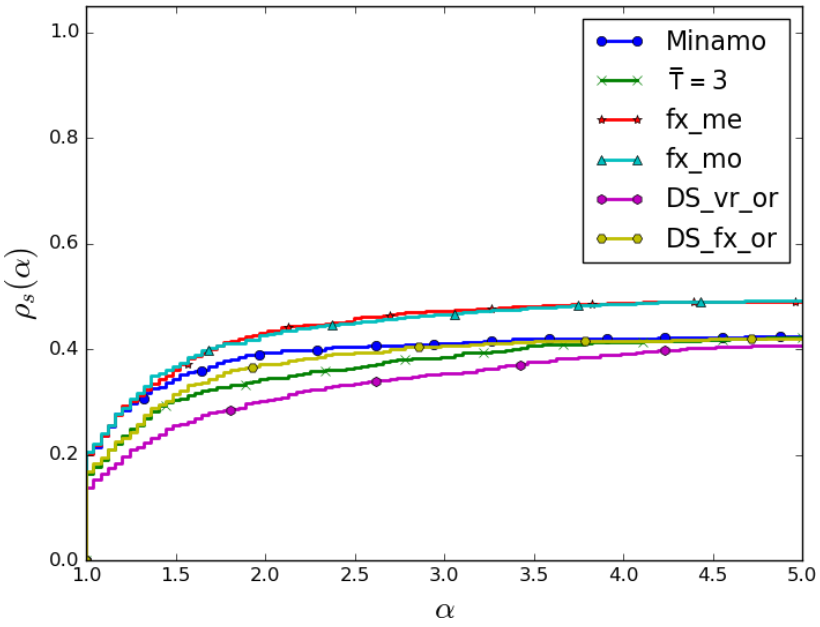


FIGURE 1 – Graphe de performance (au seuil fin) pour les meilleures méthodes du Chapitre 7 de ce mémoire.

4 Conclusion

Les deux méthodes adaptées de *FiScIS-EA* (*fx me* et *fx mo*)

— améliorent d'avantage Minamo que la version originale de *FiScIS-EA* d'un point de vue scientifique (solution finale) et d'un point de vue opérationnel (temps d'exécution) avec l'AG pur ou intégré dans une boucle *SBO* ;

— sont les meilleures méthodes de ce mémoire pour l'AG pur (solution finale et temps d'exécution) et pour l'AG intégré dans une boucle *SBO* (uniquement pour la solution finale).

Références

[1] J. COOK et D. TAURITZ, « An exploration into dynamic population sizing », *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, p. 807–814, 2010.

[2] J. MORE et S. WILD, « Benchmarking derivative-free optimization algorithms », *SIAM Journal on Optimization*, vol. 20, p. 172–191, 2009.